



M4.1

SP-DevOps concept evolution and initial plans for prototyping.

Dissemination level	RE-PU (published on 10.03.2015)
Version	1.0
Due date	31.10.2014
Version date	14.11.2014

This project is co-funded
by the European Union



Document information

Editors and Authors:

Editor: Catalin Meirosu (EAB)

Contributing Partners and Authors:

ACREO	Pontus Sköldström
BME	Felician Nemeth
DTAG	Juhoon Kim
EAB	Catalin Meirosu
iMinds	Sachin Sharma
OTE	Ioanna Papafili
POLITO	Guido Marchetto, Riccardo Sisto
SICS	Rebecca Steinert
TI	Antonio Manzalini
TUB	Nadi Sarrar, Matthias Rost, Stefan Schmid

Project Coordinator

Dr. András Császár

Ericsson Magyarország Kommunikációs Rendszerek Kft. (ETH) AB

KONYVES KALMAN KORUT 11 B EP

1097 BUDAPEST, HUNGARY

Fax: +36 (1) 437-7467

Email: andras.csaszar@ericsson.com

Project funding

7th Framework Programme

FP7-ICT-2013-11

Collaborative project

Grant Agreement No. 619609

Legal Disclaimer

The information in this document is provided 'as is', and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law.

© 2013 - 2014 by UNIFY Consortium

Table of contents

1 Introduction and relationship with other deliverables	2
1.1 Introduction	2
1.2 Summary of relevant developments in WP2	4
1.3 Summary of relevant developments in WP3	4
1.4 Summary of relevant developments in WP5	5
2 Evolution of SP-DevOps beyond the sketch in D4.1	7
2.1 High-level specification of interfaces between components in the UNIFY Architecture	7
2.1.1 Extensions to the U-SI interface	8
2.1.2 Example of interface API invocation	10
2.2 Refinement of the Observability Points	12
2.2.1 Considerations on data access and visibility	13
2.2.2 Control App and control message exchanges	14
2.2.3 Mapping of Monitoring Functions towards Universal Nodes	14
2.3 Towards a SP-DevOps Toolkit	15
3 Observability	19
3.1 Messaging system for the SP-DevOps Observability process	19
3.1.1 Measurement data and control traffic transport	19
3.1.2 Message queuing systems	20
3.1.3 Deploying a monitoring system using ZeroMQ	22
3.2 On describing monitoring functions	23
3.3 In-network aggregation with VirtuCast	25
3.4 Configuration and update consistency	27
3.4.1 State-of-the-art complement to D4.1	27
3.4.2 Contribution and First Insights	28
3.5 Capabilities and tools for the SP-DevOps Observability process	29
3.5.1 In-network link monitoring	29
3.5.2 Loss monitoring for aggregated flows	33
4 Verification	37
4.1 Deploy-time Functional Verification of Dynamic Service Graphs	37

4.1.1 NetPlumber preliminary analysis and evaluation	39
4.1.2 VeriFlow preliminary analysis and evaluation	41
4.1.3 Conclusions and planned future steps	44
4.2 Run-time Verification of Forwarding Configurations with AutoTPG	45
5 VNF Development Support	50
5.1 Multilayer, multicomponent debugging tool	50
5.2 Network watchpoints	51
6 Troubleshooting	57
7 Integrated prototyping - DevOpsPro initial plans	58
List of abbreviations and acronyms	66
8 Bibliography	67
Annex 1 SP-DevOps components for UNIFY Architecture Interfaces	70
8.1 Extensions to the SI-Or interface	70
8.2 Extensions to the Co-Rm interface	71
8.3 Extensions to the Or-Ca and Ca-Co interfaces	73
Annex 2 Mapping of progress reported towards WP4 objectives	75

List of figures

<i>Figure 1.1: SP-DevOps processes and actors in a DevOps loop [2]</i>	2
<i>Figure 1.2: Interactions between SP-DevOps processes and actors.</i>	3
<i>Figure 2.1: Representation of major interfaces in the UNIFY Architecture [3]</i>	8
<i>Figure 2.2: An overview of the mapping of MF and OPs on UNs.</i>	15
<i>Figure 2.3: SP-DevOps Toolkit in the UNIFY production environment</i>	17
<i>Figure 3.1: Clients connect to brokers using one of the supported protocols; brokers connect to each other in the same way.</i>	22
<i>Figure 3.2: Using the messaging system to connect Monitoring LCPs, CPs, and a Logger. Some components are running inside Universal Nodes and others at higher layers, e.g. in the orchestration layer.</i>	23
<i>Figure 3.3: Configuration of delay monitoring of an instantiated Firewall VNF using MEASURE and the message bus to communicate configuration, notifications, and measurement results.</i> ..	24
<i>Figure 3.4: Example state machine defined by MEASURE. Based on incoming measurements the results the state machine switches between two zones and performs actions.</i>	25
<i>Figure 3.5: Mapping of VirtuCast to an MF.</i>	26
<i>Figure 3.6: Mapping of VirtuCast to an NF-FG.</i>	27
<i>Figure 3.7: Example mapping of the link utilization monitoring function in the infrastructure layer</i>	30
<i>Figure 3.8: Example of mapped rate monitoring OPs for monitoring on Operator level</i>	31
<i>Figure 3.9: Example mapping of the link monitoring MF in the infrastructure layer.</i>	32
<i>Figure 3.10: Mapping of several delay MFs controlled by one control app for the purpose of monitoring on Operator level. Each MF consists of a set of OPs, as shown in Figure 3.7.</i>	32
<i>Figure 3.11: Example mapping of loss monitoring for aggregated flows tool as an OP</i>	33
<i>Figure 3.12: Example mapping of the loss monitoring for aggregated flows tool in the Service Graph</i>	34
<i>Figure 3.13: Loss monitoring capability inferred in the infrastructure management and represented as NF-FG</i>	36
<i>Figure 4.1: NetPlumber Workflow</i>	40
<i>Figure 4.2: NetPlumber performance in our testbed.</i>	41
<i>Figure 4.3: VeriFlow Equivalence Classes generation process.</i>	42
<i>Figure 4.4: The VeriFlow UML class diagram.</i>	43
<i>Figure 4.5: VeriFlow TCP connection setup latency.</i>	44
<i>Figure 4.6: Matching header failure detection.</i>	46

<i>Figure 4.7: Example mapping of the AutoTPG tool in the infrastructure layer/virtualization layer</i>	<i>48</i>
<i>Figure 4.8: An NF-FG mapping for the AutoTPG tool.....</i>	<i>48</i>
<i>Figure 5.1: Schematic architecture of the watchpoint tool.....</i>	<i>52</i>
<i>Figure 5.2: Structure of network watchpoint</i>	<i>53</i>
<i>Figure 5.3: Use-case 1 (explicitly defined control app).....</i>	<i>55</i>
<i>Figure 5.4: Use-case 2 (without explicitly defined control app)</i>	<i>56</i>

List of tables

<i>Table 1: Supported programming languages for various messaging platforms.....</i>	<i>21</i>
<i>Table 2: Short summary of progress reported in this milestone mapped towards DoW objectives</i>	<i>75</i>

Summary

This milestone document reports on WP4 progress after the submission of the D4.1 deliverable in August 2014. The Service Provider DevOps concept (SP-DevOps) was introduced in D4.1 as a set of processes that apply principles inherited from the data centre DevOps movement to the UNIFY production environment. In this milestone, it is further integrated into the UNIFY Architecture by presenting initial APIs that incorporate functions and data originating from components related to Observability, Verification and Virtual Network Function (VNF) Developer support processes throughout the UNIFY Architecture. Furthermore, the milestone includes a first description of the SP-DevOps Toolkit, which complements the APIs and enables a close integration between capabilities developed in WP3, WP4 and WP5 with the aim of simplifying the management of software-defined telecom network infrastructure. The Observability Points, already outlined in D4.1, are further refined in this document by addressing problems in key areas such as configuration consistency and scalability of the messaging component. We provide a first report on prototyping and evaluation activities undertaken by the partners in the Workpackage. Although such activities were performed individually while this document was prepared, initial plans for integration of partner work are presented in the document.

The manuscript is organized as follows. Section 1 summarizes the main SP-DevOps components from D4.1, as well as relevant developments from WP3 and WP5. Section 2 presents the evolution of the SP-DevOps initial concept. APIs between components of the UNIFY Architecture are described partly in this section and partly in Annex 1 to avoid repetitions. The Observability Points are refined in terms of architectural mapping towards the UNIFY Universal Node. The SP-DevOps Toolkit is outlined, bringing together annotations associated to monitoring and verification of the Network Function Forwarding Graphs (NF-FGs) and a set of tools developed in WP4 to address specific research challenges identified in D4.1. Section 3 details the progress towards realizing components of the SP-DevOps Observability process, touching upon the consistent configuration and scalability of messaging for Observability Points as well as several novel tools designed for estimating packet delays and losses in OpenFlow networks. Section 4 reports on our investigations on the potential for extending state-of-the-art OpenFlow formal verification tools to the UNIFY production environment, as well as presenting for the first time AutoTPG, a new tool designed to address data plane verification problems in a more scalable and comprehensive way compared to existing solutions. Section 5 outlines two components that were designed to address challenges related to the SP-DevOps VNF Development Support process, namely an extension to the popular Emacs editor for launching a software debugger on a VNF deployed in a NF-FG and a watchpoint tool that transparently inspects the OpenFlow control plane to provide insights for the developers. The very short Section 6 exposes the relationship between the SP-DevOps Troubleshooting process and two of the tools already presented. Section 7 outlines per-partner prototyping plans as well as first ideas on candidate functionality for integration in DevOpsPro. Annex 2 maps the progress reported in this document versus the Workpackage objectives from the Description of Work.

1 Introduction and relationship with other deliverables

The UNIFY project aims at building a framework that enables the agile and flexible creation of Internet services. The successful development of such a framework will eventually lead telecom providers to the innovative, efficient, and cost-effective operation of services. To accomplish this goal, the UNIFY consortium is working on realizing important features including virtualization of network functions, coordination of resources, and automated management of service elements. Recently emerged paradigms, e.g., Software Defined Network (SDN), Network Function Virtualization (NFV), and cloud computing, have shown that their capabilities meet the high level requirements for developing the aforementioned features, thus these paradigms are chosen as cornerstones of the UNIFY framework.

1.1 Introduction

In the UNIFY project, the overall goal of WP4 is to develop methods that integrate service deployment with service operation steps in a software-defined telecom infrastructure. The focus areas include service validation, monitoring, and troubleshooting. The bringing together of modern agile software development and operations methods is commonly referred to as DevOps. Developing appropriate tools and processes is an important activity to reflect the major principles of DevOps, i.e. monitoring and validating operational quality, developing and testing against production-like systems, deploying with repeatable, reliable processes, amplifying feedback loops. Base requirements for applying DevOps to software-defined service provider infrastructure were defined in D4.1 [1], along with the base constructs associated to the SP-DevOps concept as presented in *Figure 1.1*.

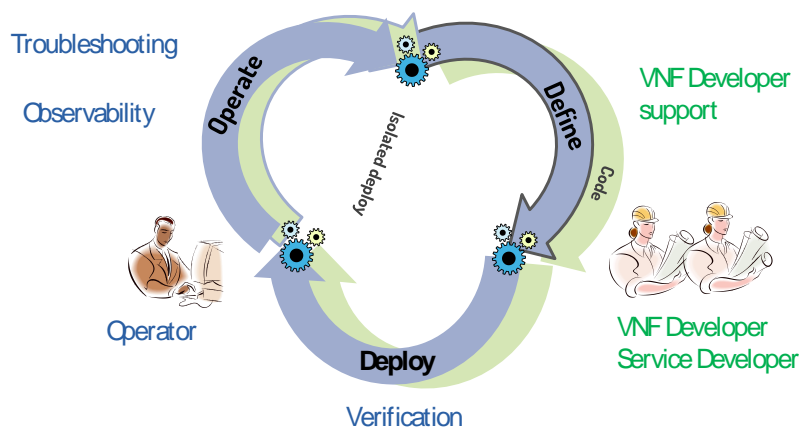


Figure 1.1: SP-DevOps processes and actors in a DevOps loop [2]

Three distinct, although closely related, roles were defined: the Service Developers and VNF Developers that define or write the code for a new service or a new Virtual Network Function (VNF) that need to be deployed in the UNIFY production environment, and the Operator, who perform the daily operations tasks associated with the service and supporting VNFs in interaction with the relevant Developers. Four processes were defined and mapped towards components in

the UNIFY Functional Architecture: Observability, Verification, Troubleshooting, VNF Development Support. The interactions between the processes and the SP-DevOps actors are summarized in *Figure 1.2*.

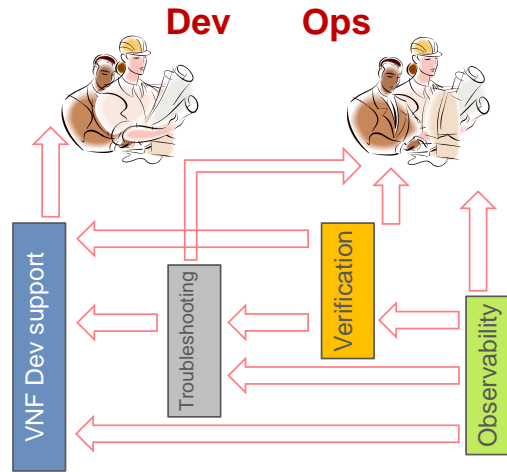


Figure 1.2: Interactions between SP-DevOps processes and actors

The arrows in *Figure 1.2* represent the major directions of information flows between SP-DevOps processes and actors. The Observability process provides the raw information to the other processes, in terms of values of metrics associated to the UNIFY compute, network and storage resources. The Verification process covers two aspects, a formal verification of properties of the Network Function - Forwarding Graphs (NF-FGs) prior or during deployment, and a verification of network forwarding configuration while the service graph is executed in the production environment which may include application-specific configurations (e.g., firewall policies). The VNF Development Support process provides additional visibility into the software-defined infrastructure as well as the capability to control in a fine-granular manner the placement of certain Observability Points. Within the Troubleshooting process, Operators and Developers use data generated by the Observability and Verification processes that are automatically aggregated in workflows to investigate anomalies in the production environment.

D4.1 also presented initial views on Monitoring Functions (MFs) as VNFs that implement novel observability capabilities in the UNIFY production environment. Observability Points (OPs) were defined to implement measurement and modelling functionality local to a virtual node and consist of two types of components operating in the local Control Plane and the Data Plane of that node.

This milestone is provided as an evolution of the SP-DevOps sketch depicted in [1]. We describe interfaces that integrate SP-DevOps functionality with the UNIFY architecture. An initial view of SP-DevOps Toolkit is presented, along with details on the progress made implementing tools that support it. Finally, a first overview is given on agile evaluation and prototyping activities along with efforts to align towards an integrated prototype.

In order to align the development of SP-DevOps with the requirements of the overall UNIFY architecture, it is crucial to address changes and progress made in other work packages. To this end, the recent development of other work packages is summarized in the remaining part of this section.

1.2 Summary of relevant developments in WP2

WP2 defines use cases and the architecture of the UNIFY framework based on the requirements from stakeholders. Furthermore, it plans to integrate prototypes developed from WP3 (Service Programming, Orchestration and Optimization), WP4 (Advanced Management Framework and Tools), and WP5 (Universal Node Architecture and Evaluation).

WP2 released Deliverable 2.1 [2]. D2.1 to define the key design principles of the UNIFY framework and provide the initial proposal of the UNIFY Architecture, including the identification of 46 requirements collected from all the Workpackages, and definition of 11 use cases. Furthermore, D2.1 identified the fundamental processes supported by the UNIFY Architecture, i.e., boot-strapping, programmability, verification, observability, troubleshooting, and VNF development. The initial UNIFY Architecture was described as two steps, i.e., an Overarching Architecture and a Functional Architecture.

Deliverable 2.2 [3], to be released simultaneously with this milestone, revisited the architecture with the purpose of identifying further reference points and clarifying the relations between components in the refined architecture. Major aspects included revising the Cf-Or interface (*Figure 2.1*) for the interaction between the UNIFY Orchestration Layer and resource control components of VNFs and the inclusion of the *Virtualizer* components to take care of isolation and resource abstraction at the NF-FG levels. The UNIFY Functional Architecture, not included as a figure in this milestone but presented in [3], includes two major functional blocks related to SP-DevOps termed *Observability and Performance Manager* and *Verification Manager*. The exact mapping of SP-DevOps processes and capabilities towards these functional blocks in the UNIFY Architecture will be detailed in the next WP4 milestone document. We note nevertheless that these components already existed in the version of the UNIFY Functional Architecture that we used for mapping the SP-DevOps processes documented in D4.1, so no major changes are to be expected in this respect.

1.3 Summary of relevant developments in WP3

Deliverable D3.1 [4], also to be released simultaneously with this milestone, presents the major components of the UNIFY Programmability Framework. The programmability aspects focus on provisioning resources through the reference points of the UNIFY Architecture, the information models associated to them and a set of function calls that transfer the information.

The description of the information model for the NF-FG is of particular relevance to WP4. The core primitives of this model are endpoints, Network Functions, network elements and monitoring parameters. The monitoring parameters could be measured through standards-

defined counters or standards-based Operations, Administration and Maintenance (OAM) tools, or some of them could be measured using SP-DevOps MFs and OPs. When included in an NF-FG, MFs are to be represented using the Network Function primitive. D3.1 introduced a visual representation for these primitives, as well as an in-depth specification of the elements of the NF-FG model. In this document, we will use the same visual representation for depicting ways of integrating particular types of MFs that result out of WP4 work in a NF-FG. D3.1 discusses the integration of monitoring in the Programmability Framework as annotations to the NF-FG through joint work with WP4, and gives an overview of the MEASURE language that this milestone mentions in section 3.2.

D3.1 also presented initial definitions of interfaces between the separate layers of the UNIFY Architecture, based on the reference points and the information models presented in the document. A set of operations is related to data that could be obtained through SP-DevOps Observability processes – for example, the *service report*, *get Service Graph info*, *notification / alarm* on the U-SI interface, *get / send observability info* and *notification / alarm* on the SI-Or interface. In this milestone, section 2.1 and Annex 1 present in further detail how such operations could be implemented through a set of API calls that ensure that are designed to be compliant with the recursiveness property of the UNIFY Architecture.

The process and information flows detailed for the use cases presented in Section 8.1.4 of D3.1 showed how data obtained from the SP-DevOps Observability process could be used within the UNIFY Orchestration layer. In particular, four of the use cases described the information flow for triggering a scale-up, scale-down, scale-in or scale-out of resources allocated to a NF-FG based on monitoring information.

D3.1 approached the problem of defining the differences between mapping functionality that is an inherent part of the Resource Orchestrator and verification functions aligned with the SP-DevOps Verification process. It defined a verification role as validating, against a set of policies, the embedding that was found by the mapping functionality. In a distributed orchestration solution, the verification may be necessary for checking whether the resources promised are still available once the distributed mapping process was finalized. The verification is also relevant as a step before the scoping process in the Resource Orchestrator, allowing proving certain constraints for topological correctness (such as reachability of the resources, absence of loops).

1.4 Summary of relevant developments in WP5

Deliverable D5.2 [5] presented the architecture of a Universal Node (UN) and detailed its main functional components: the Host Environment, the Unified Resource Manager, the VNF Execution Environment and the Virtual Switching Engine. SP-DevOps OPs and MFs require access to compute, network and storage resources from the UN, and thus the Unified Resource Manager is a key component, as it has complete understanding over the resources available. The VNF Execution Environment provides the setting where OPs and MFs will execute their Control Plane functionality. D5.2 specified the VNF Template and Image Repository interfaces, leveraging the

HTTP protocol and JSON formal to implement a RESTful API. On the longer term and as soon as a reference implementation becomes available, WP4 needs to steer the contributions towards the DevOpsPro prototype towards supporting these interfaces in order to facilitate the integration of functionality in the project-wide prototyping effort.

D5.2 identified several types of Virtual Network Functions that could be supported by the VNF Execution Environment. *VNF Type 2s* are isolated containers running on the same operating system kernel as the host. *VNF Type 3s* are additional processes running on the host operating system alongside the other components of the UN such as the virtual switching engine. *VNF Type 4s* are plugins to the Virtual Switching Engine. We observe that these VNF types fulfil the node-level requirements defined in D4.1 for realizing the SP-DevOps concept in UNIFY and are appropriate for implementing different types of MFs and OPs. For example, OPs that require stringent Data Plane (DP) interaction such as additional OpenFlow actions in case the DP is OpenFlow-compliant may be implemented as *VNF Type 4* plugins to the Virtual Switching Engine or embedded within it should the implementation not offer the modularity required for a plugin. OPs that require access to existing counters but aggregate the information at the node or port levels could be implemented as *VNF Type 3s* or *VNF Type 4s*. Tools that generate packets to test DP properties such as reachability or consistency could be implemented as *VNF Type 2* containers.

2 Evolution of SP-DevOps beyond the sketch in D4.1

2.1 High-level specification of interfaces between components in the UNIFY Architecture

This section defines additions to the high-level specification of the interfaces between components of the UNIFY Architecture. The additions are made in order to support the four SP-DevOps processes (Observability, Verification, Troubleshooting, and VNF Support) that were introduced in D4.1 [1].

Deliverable D2.2 [3] defined the set of interfaces between the high-level components of the UNIFY Architecture as presented in *Figure 2.1*. The major interfaces are U-SI between the Application Layer and the Service Layer, SI-Or between the Service Layer and the Orchestration Layer, Co-Rm between the Orchestration Layer and the Infrastructure Layer. Two other interfaces, Or-Ca and Ca-Co, are of interest for the operations of the infrastructure and for the developers. We consider that the Cf-Or interface between the Resource Orchestration Layer and the VNF Control Plane App is a separate case in terms of requirements for observability and verification that will be addressed in the next milestone.

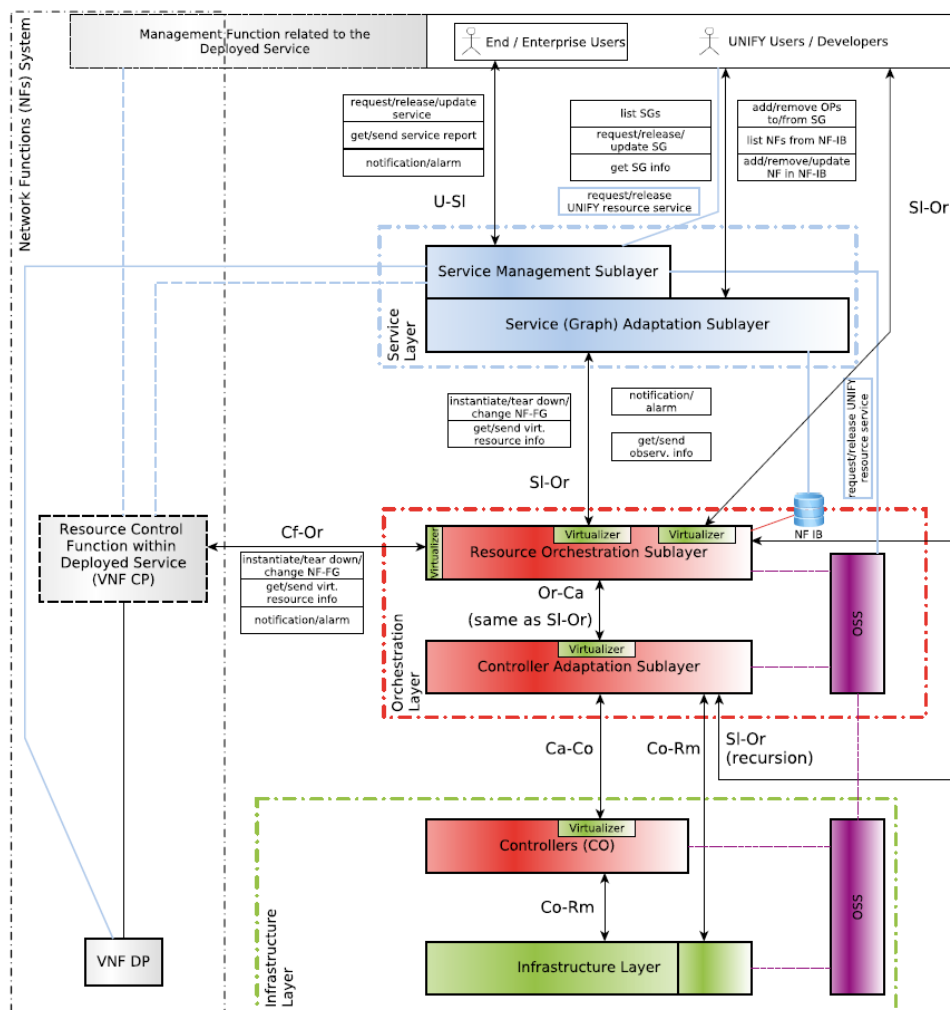


Figure 2.1: Representation of major interfaces in the UNIFY Architecture [3]

Recent work in ETSI started to define interfaces between components of the NFV-MANO architecture [6]. We use the latest public version of the document (0.6.1, July 2014) available at the time of writing as inspiration for interface component regarding monitoring. We also took inspiration from the TMForum Simple Management API [7], as well as from the ITU-T Y.1654 Ethernet service activation specification. All these specifications aim to support carrier-grade features, and therefore we wanted to ensure that the interfaces we define can provide an equivalent level of features for service graphs. The function calls to be supported through these interfaces are very similar in terms of number of parameters and syntax. It was designed in order to support the recursiveness property of the UNIFY architecture. This feature is also common to modern programming languages where it is known as polymorphism. The extensions for the U-SI interface are included in section 2.1.1 below, while the other interfaces are detailed in Annex 1. Section 2.1.2 includes two examples of API invocation, traversing the APIs at all the interface levels.

2.1.1 Extensions to the U-SI interface

The following functions are proposed to be implemented through the U-SI interface in order to support the SP-DevOps Observability, Verification, Troubleshooting and VNF Development Support processes:

GetExecutionState - returns an object or a set of objects providing information about the execution state of the Service Graph or to particular components. The function is called an entity in the Service Developer or VNF Developer role and the result is returned by the Adaptation Functions in the UNIFY architecture from *Figure 2.1*

Input Parameters: Service Graph identifier (mandatory; the data type of the parameter is the same as the NF-FG Id specified in [4]), list of Service Graph component identifiers (optional)

Output Parameters: set of execution states associated to the Service Graph Identifier or the Service Graph component identifiers. Example of execution states include: deployed, activated, running, stopped, debugging, suspended, decommissioned. The exact meaning of the execution states is left for definition at a later stage.

SetExecutionState -changes the execution state of the service in order to allow a Service Developer or a VNF Developer to perform troubleshooting or debugging activities.

Input Parameters: Service Graph identifier (mandatory), desired execution state (mandatory)

Output Parameters: Boolean variable indicating whether the transition could be applied (true) or not (false)

GetPerformanceValues - collects the latest values measured for a list of performance metrics associated to the Service Graph

Input Parameters: Service Graph identifier (mandatory), list of metrics for which the values are to be returned (optional)

Output Parameters: Set of metric-value pairs measured during the last interval for all the metrics associated with the Service Graph (in case a list of metrics was not provided) or only for those metrics specified in the optional input parameter. Different metrics might have different time values for the last interval; this information is expected to be included in the metric description metadata.

Verify - performs one or more verification actions on some or all the verifiable properties associated with a Service Graph

Input Parameters: Service Graph identifier (mandatory), list of properties that need to be verified (optional), verbosity level (Optional). Such list of properties that could be verified include packet transit through the Service Graph, reachability of various Service Graph components, the absence of loops or the maximum number of times a particular component of a Service Graph is to be traversed, etc. A comprehensive list of properties that need to be verified is expected to be dependent on the Service Graph, and include a description of the VNFs that are part of the graph as well as actual configuration parameters of the VNF instance in case they need to be verified. The verbosity level is an optional parameter that controls the detail included in the results. For example, with a verbosity level set to "low", a verification of the absence of loops could return simply "pass" or "fail", while the same verification action could in addition return the identifiers of the Service Graph components that are part of the detected loop.

Output Parameters: list of results from the verification actions performed on the service.

SetupNotification - configures automated notifications regarding performance metrics or state transitions associated with the Service Graph

Input Parameters: Service Graph identifier (mandatory), listener identifier (mandatory), list of metrics or state transitions for which the values are to be returned (optional), notification criteria (optional). The listener is an identifier for an entity that receives the notification. The listener must be registered with the RegisterListener call beforehand. The notification criteria allow specifying, within certain limits, filtering capabilities for the issuing of notifications. For example, a notification could be sent only for a pre-determined number of times within a pre-defined time window, or be generated only when the value of a particular metric rises above or descends below a certain threshold (hence no continuous updates are generated at pre-defined time intervals), etc.

Output Parameters: Boolean value reflecting whether the setup was successful (True) or not (False).

Notify - provides automated notifications regarding performance metrics or state transitions associated with the Service Graph as configured via the SetupNotification function

Input Parameters: None

Output Parameters: Service Graph identifier (mandatory), metric or state identifier (mandatory), Type-Length-Value (TLV) triplet representing the information to be notified to the listener(s).

RegisterListener / UnregisterListener - makes a change to the Service Graph to include a communication endpoint with an entity other than the Service Developer or the Service Owner (customer) such that this entity is able to invoke the GetPerformanceValues or Verify function calls on the interface and receive the results, or receive notifications through the Notify function. Such entities could be, for example, a ControlApp associated with functions such as scaling in the UNIFY Orchestration, or an OSS of the service provider that performs customer experience management functions.

Input Parameters: Service Graph identifier (mandatory), Service Graph components for adding or removing an additional communication point. The Service Graph components include the NF-FG description of resources that need to be allocated for the communication point.

Output Parameters: Boolean value that indicates whether the operation was successful (true) or not (false)

2.1.2 Example of interface API invocation

In this subsection we provide, as a way of illustrating potential uses of these generic interfaces, two possible examples that reflect how the API could be invoked. Depending on the actual implementation, other call sequences might be possible as well.

We assume that a Service Developer would need to obtain on demand the instantaneous value of the CPU utilization of the container where one of the instances of an Elastic Router [d2.1] was deployed. We also assume that the ElasticRouter ControlApp asked to be notified by the UNIFY production environment when new information for this metric becomes available by invoking a RegisterListener call on the Cf-Or interface during its bootstrap process. This is an example of a sequence of API calls that could be triggered through the interfaces in order to provide this value (other sequences might be possible; the sequence of calls below is provided for illustration only and does not mean that an SP-DevOps prototype must exhibit this exact call sequence):

U-SI: GetPerformanceValues("MyServiceGraph", "ElasticRouter-CPU usage")

In turn, this could generate the following calls in the UNIFY production environment (the changes in the parameters used follow the level of resources abstraction managed at each layer):

SI-Or: GetPerformanceValues("MyNFFG", "ElasticRouter-CPU usage")

Or-Ca: GetPerformanceValues("MyNFFG-ElasticRouter", "CPU usage")

Ca-Co: GetPerformanceValues("MyNFFG-ElasticRouter", "CPU usage")

*Co-Rm: GetPerformanceValues("VirtualCPU1", "CPU usage")
returns "CPU usage"=50%*

*Co-Rm: GetPerformanceValues("VirtualCPU2", "CPU usage")
returns "CPU usage"=75%*

Ca-Co: returns "VirtualCPU1=50%" and "VirtualCPU2=75%"

Or-Ca: returns "MyVirtualCPU1=50%" and "MyVirtualCPU2=75%"

SI-Or: returns "ElasticRouterCPU1=50%" and "ElasticRouterCPU2=75%"

U-SI: Notify "MyNFFG", "ElasticRouterCPU1=50%" and "ElasticRouterCPU2=75%"

In the second example, we consider a situation when a VNF Developer needs to investigate a problem with an ElasticRouter that does not seem to forward enough traffic according to the capacity specified in the Service Graph. The problem is identified as a misconfiguration in one of the virtual switches associated with the service. We expect that only a VNF Developer or Operator role has access to this level of detail regarding infrastructure resources. Nevertheless, the VNF Developer is expected to access primarily properties that are derived from the actual configuration of the forwarding resources, rather than the actual configuration itself. An example of a sequence of API calls that could be triggered through the UNIFY interfaces in order to investigate a problem is presented further down in this section. Other sequences might be possible; the sequence of calls below is provided for illustration only and does not mean that an SP-DevOps prototype must exhibit this exact call sequence.

VNF Developer *Us-SI: SetExecutionState("MyServiceGraph", "debug")*

U-SI: returns OK

VNF Developer *Us-SI: Verify("MyServiceGraph", "property=reachability", "verbosity=low")*

Or-Ca: Verify ("MyNFFG-ElasticRouter", "property=reachability", "verbosity=low")

Ca-Co: Verify ("MyNFFG-ElasticRouter", "property=reachability", "verbosity=low")

Co-Rm: Verify ("FlowDescrVSwitch1, FlowDescrVSwitch2, FlowDescrVSwitch3", "property=reachability", "verbosity=low") returns Fail

Ca-Co: returns Fail

Or-Ca: returns Fail

U-SI: returns Fail

VNF Developer *U-SI: Verify("MyServiceGraph", "property=reachability", "verbosity=high")*

Or-Ca: Verify ("MyNFFG-ElasticRouter", "property=reachability", "verbosity=high")

Ca-Co: Verify ("MyNFFG-ElasticRouter", "property=reachability", "verbosity=high")

Co-Rm: Verify ("FlowDescrVSwitch1, FlowDescrVSwitch2", "FlowDescrVSwitch3", "property=reachability", "verbosity=high")

Co-Rm: returns "FlowDescrVSwitch1=OK, FlowDescrVSwitch2=Fail FlowDescrVSwitch3=OK"

Ca-Co: returns MyNFFG-FlowDescrVSwitch1=OK, MyNFFG-FlowDescrVSwitch2=Fail MyNFFG-FlowDescrVSwitch3=OK"

Or-Ca: returns MyNFFG-FlowDescrVSwitch1=OK, MyNFFG-FlowDescrVSwitch2=Fail MyNFFG-FlowDescrVSwitch3=OK"

U-SI: returns "reachability fail, cause = MyNFFG-FlowDescrVSwitch2"

2.2 Refinement of the Observability Points

Observability Points (OPs) were defined in D4.1 as containers for observability capabilities. As such capabilities are relevant to all four SP-DevOps processes (Observability, Troubleshooting, Verification and VNF Development Support), we present generic aspects related to OPs as part of the evolution of the integrated SP-DevOps concept. The aspects covered encompass monitoring roles, control of OPs and initial mapping of Monitoring Functions (MF) to the architecture. Additional considerations regarding messaging, configuration and consistency will be made in Section 3, because they include insights from ongoing technology evaluations that are not mature enough to be considered as integration aspects that affect the specific capabilities developed by different partners. The refinement of the OPs as presented in this section is the result of cooperation with WP3 and WP5.

2.2.1 Considerations on data access and visibility

The SP-DevOps concept [1] defined three roles that are of interest when applying DevOps principles to a telecom infrastructure environment: the Service Developer, VNF Developer and the Operator. The following aspects are further identified regarding permissions to access monitoring and troubleshooting capabilities, data and information about the underlying infrastructure that is to be used entities having the three SP-DevOps roles:

1. Infrastructure (Operator, VNF Developer) - physical resource (link, node) status
2. Service/NF-FG (Operator, VNF Developer) - logical E2E status, root cause analysis
3. Service/NF-FG (Service Developer) - logical E2E, SLA monitoring
4. As a service/NF-FG (Service Developer, VNF Developer) - logical, monitoring as VNF

The different aspects control the level of detail provided by a deployed MF that is accessible to an Operator or a Developer. For example, a deployed MF observing the performance in a certain part of a NF-FG, will reveal to an Operator the exact location in the infrastructure where a performance degradation has been detected, but will on a Service Developer monitoring level only notify about the existence of the performance degradation without specifying the location.

The *infrastructure monitoring* aspect is not necessarily tied to the topology of an NF-FG, and includes monitoring of physical links (e.g. in terms of metrics such as delay, jitter, loss, and utilization) and compute resources (status, utilization). In general, monitoring on infrastructure level is primarily used for: assisting the resource orchestration by providing resource availability information; troubleshooting and root cause analysis; and, fault and performance management (e.g. fast detection of physical failures and failover strategies). The *service/NF-FG monitoring on Operator or VNF Developer* aspect includes monitoring of similar metrics as on the infrastructure level monitoring, but is performed for virtual resources and often requires low-level access to the infrastructure equipment for e.g. end-to-end performance measurements, SLA monitoring, troubleshooting, and fault management for logical topologies.

On Service Developer level monitoring we identify the aspects of *service/NF-FG monitoring* and *NF-FG monitoring as a service*. *Service/NF-FG monitoring* is used for monitoring of individual services and logical entities for testing e.g. the logical end-to-end or segment delay, loss, jitter, and utilization. It is applied for the purpose of monitoring SLAs, resource management support (such as triggering re-scaling actions) and for root cause analysis at the Service Graph level. There is no or very restricted access to data from the underlying virtual infrastructure. Finally, the *NF-FG monitoring as a service* aspect encompasses the highest level of monitoring where the MFs are deployed as VNFs, meaning that the placement of OPs and allocation of MFs is part of the decomposition processes, and that the access to monitoring information is restricted only to logical links. A typical use case for such a VNF MF would be a virtual DPI function for controlling and testing the processing of packets before and after a VNF in a NF-FG.

2.2.2 Control App and control message exchanges

The control component in the UNIFY Universal Node Local Resource Managers (LRM) as described in D4.1 is identical with a Control App as specified in D5.2 [5] but implements MF-specific functionality, meaning that it:

- manages one or several OPs within the scope of an MF;
- manages one or several MFs;
- aggregates results;
- forwards reports/internal control messages to upper layers of the architecture or between other control apps.

A Control App manages one or several OPs per MF, but can also be assigned to manage several MFs within an NF-FG. The Control App keeps track of the individual OP instances and corresponding identifiers within and between MFs for direct communication between OPs where applicable. Information exchange and analytics including MFs managed by different Control Apps (for the purpose of e.g. troubleshooting) cover communication between Control Apps located at levels parallel with the LRM and higher layers of the UNIFY Architecture.

2.2.3 Mapping of Monitoring Functions towards Universal Nodes

Monitoring of a NF-FG is performed by the means of one or several MFs under the control of one or several Control Apps. The functional scope of a MF typically covers one or several OPs deployed in the virtual infrastructure. Depending on the type of MF, implementation of OP capabilities includes measurement mechanisms, aggregation and analytics, as well as communication between OPs. *Figure 2.2* illustrates how an OP could be mapped to different components in the UNIFY Universal Node that was described in D5.2 [5]:

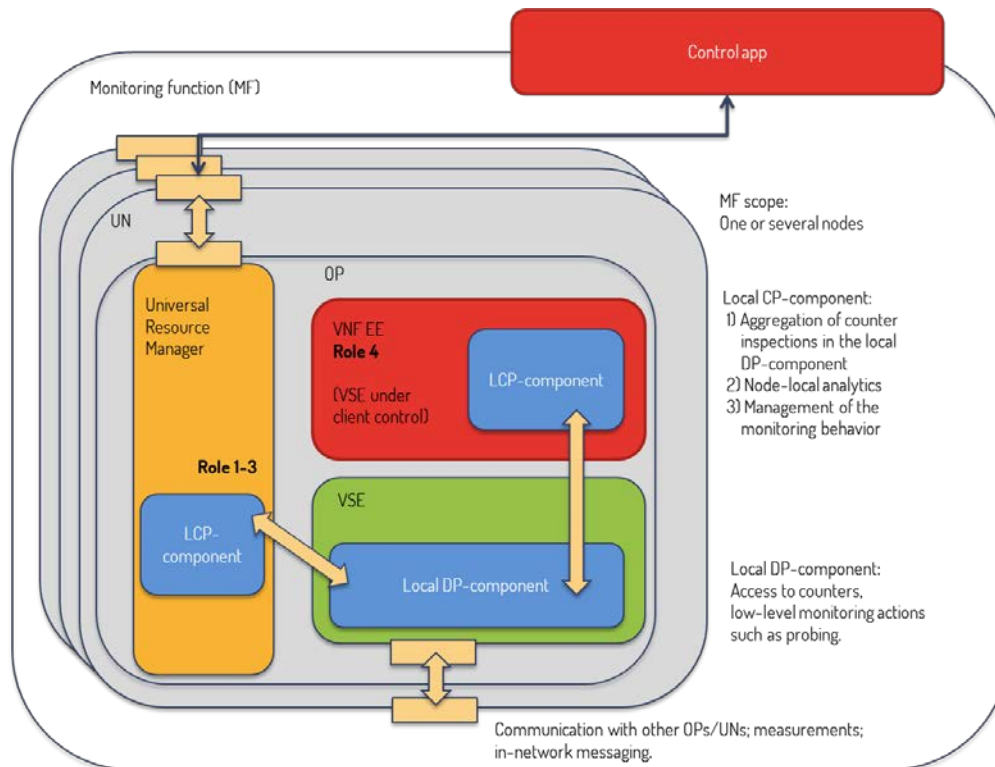


Figure 2.2: An overview of the mapping of MF and OPs on UNs.

Depending on whether the MF is instantiated on Operator level (aspect 1-3) or Service/VNF Developer level (aspect 4 in Section 2.2.1), the Local Control Plane (LCP) responsible for local aggregation and modelling in the Universal Resource Manager (URM) would mainly run in as part of the VSE manager or in the VNF Execution Environment (EE), using aggregated input and counter inspections from the local datapath component (which may be e.g. an Openflow agent). When the MF is operating on Operator level, the LCP is locally managed by the VSE manager in the URM, and implements the monitoring behaviour and node-local analytics based on input from the LDP-component. When the MF is operating on Service/VNF Developer level, the LCP implements monitoring functions and node-local analytics in the VNF EE. The Control App is responsible for managing the OPs within a MF in terms of configuration and placement as well as for forwarding reports and notifications to other parts of the architecture. Additionally, OPs can communicate via the existing interfaces for communication between UNs.

2.3 Towards a SP-DevOps Toolkit

The analysis of the DevOps movement in terms of impact on the network field that was documented in D4.1 outlined a number of tools, primarily targeting configuration management tasks, as a major part of the arsenal that developers and operators need to master. A recent article [8] provides a detailed overview on how equipment manufacturers integrate their equipment with data-centre originating tools such as Chef and Puppet. Each tool has its own interface, and each tool is individually integrated in the environment (potentially, operating in parallel with an SDN controller on the same resources, for example) in order to address the

diversity of personnel expertise of enterprise customers. While providing a good degree of flexibility, this approach is open to operations conflicts between different tools controlling different parameters at the same time, consumes resources at runtime for keeping the interfaces functional and makes the equipment vulnerable to changes introduced by third-party tool that are outside of the quality control zone of the manufacturer.

A recent blog article argued that “Enterprise DevOps” is not a new form of DevOps [9]. The authors define DevOps as “the activity of optimizing the development-to-operations value stream by creating an increasingly smooth, fast flow of application changes from development into operations, with little waste. Optimization of the value stream takes place continuously using various continuous improvement techniques” and conclude that “If you’re a large enterprise, the problems and constraints that you face will undoubtedly be different than those of an agile web startup. So, you’ll necessarily need to select different solutions. [...] Rather, it means that you’re doing DevOps correctly.” D4.1 argued for the need of a SP-DevOps concept as an interpretation of generic DevOps to address challenges that are specific to telecom service provider environments. By extending the arguments from [9] to telecom infrastructure, SP-DevOps might not be a new form of DevOps as such, but it must include solutions that are uniquely tailored for its environment. In this milestone, we introduce the SP-DevOps Toolkit as a way of describing a set of DevOps solutions that are developed targeting research challenges identified in the UNIFY production environment.

In D4.1, we introduced the four generic processes that support Service Developers, VNF Developers and Operators in a UNIFY production environment and outlined the components of the UNIFY functional architecture that are involved in supporting these processes. In section 2.1 of this milestone we presented a high-level specification of function calls over interfaces in the UNIFY architecture that act as enablers for the four SP-DevOps processes. This section outlines the components of the SP-DevOps Toolkit, which is an assembly of integrated tools and templates supporting personnel fulfilling one of the three roles (Service Developers, VNF Developers and Operators) defined in SP-DevOps.

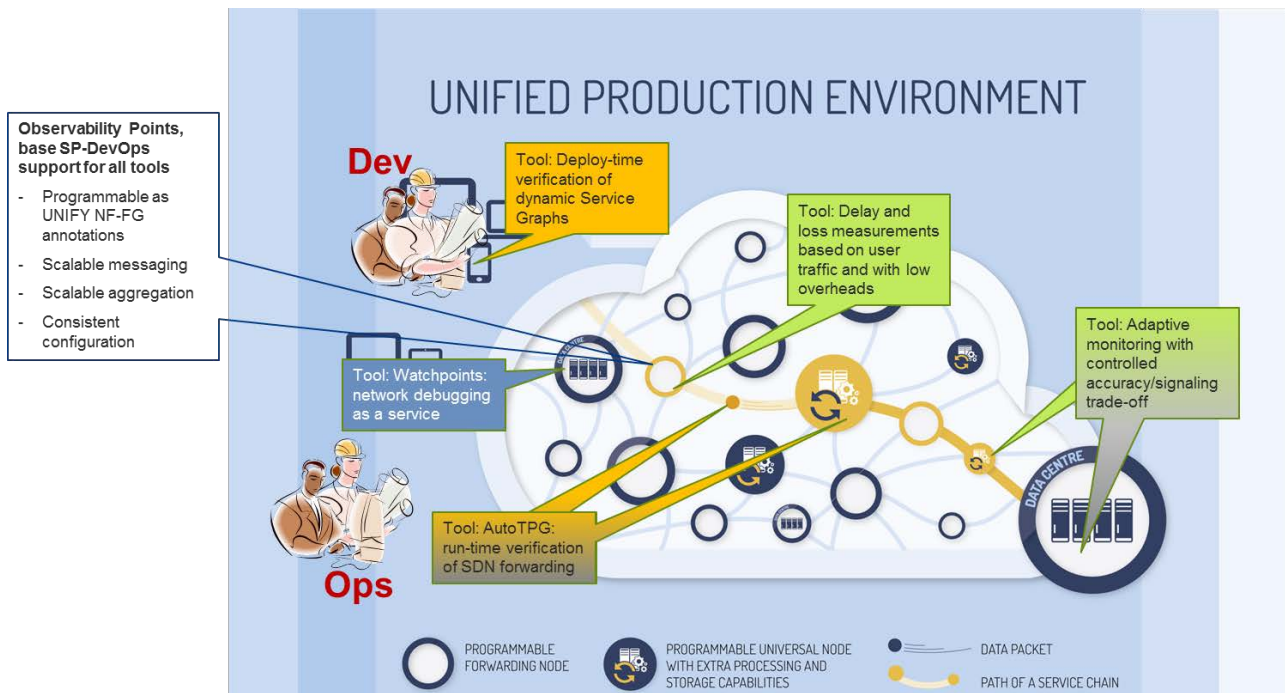


Figure 2.3: SP-DevOps Toolkit in the UNIFY production environment

The initial SP-DevOps Toolkit is represented in Figure 2.3 and will have the following components:

- NF-FG template annotations describing, in a formal language, performance metrics to be monitored as well as functions that implement the monitoring functionality as Observability Points. Work in progress in this direction is reported in Section 3.2
- NF-FG template annotations describing, in a formal language, properties to be verified as well as functions that implement the verification functionality. Work is in very initial stages, and it is not reported in this document.
- An Observability Point description and reference implementation, including interfaces to integrate with the UNIFY production environment, and capabilities for scalable information dissemination (marked as “Scalable messaging” and “Scalable aggregation” in Figure 2.3) and consistent configuration. Work in progress towards realizing the Observability Point is outlined in sections 2.2, 3.3, 3.4
- A set of tools addressing very specific problems in areas of high relevant for developing and operating software-defined infrastructure in telecom networks. Each of the tools in the SP-DevOps toolkit will be accompanied by a template description and examples on how to integrate it with NF-FGs derived from UNIFY use cases in joint work with the other Workpackages. Where relevant, the tools will follow the Observability Point reference implementation regarding message dissemination, aggregation and configuration consistency. In this document, we report on initial versions for a number of tools. In general, the work on the tools is progressing in parallel with the work supporting

generic capabilities for Observability Points. Statistical counters as outlined in 3.5.1 provide a better visibility onto fine-grained evolution of performance metrics related to ongoing traffic and are part of the adaptive monitoring shown in *Figure 2.3.*. The loss monitoring tool described in section 3.5.2 provides visibility onto the performance of aggregated network flows. The AutoTPG tool in section 4.2 has detailed verification capabilities on a host of properties related both to NF-FGs and traffic forwarding between VNFs. The watchpoint described in 5.2 checks the often overlooked OpenFlow protocol control plane for inconsistencies and alerts the controller when anomalies are detected. An investigation on existing SDN verification tools is presented in section 4.1 as a stepping stone towards verifying dynamic Service Graphs.

3 Observability

This section details progress on defining capabilities related to research challenges we identified in the Observability area and documented in D4.1. The section starts with work describing work on a messaging system, description language as well as configuration consistency techniques that could be employed by MFs and OPs in general, regardless on the actual functionality or the project partner that worked to address the specific questions solved by a particular OP or MF. We then describe progress made by individual partners on tools that will provide specific measurement capabilities (for metrics such as delay and loss) needed by the SP-DevOps Observability process. The integration of each tool in the project framework is explained through a representing their resources in NF-FGs, as well as a mapping to the OP architecture evolution detailed in Section 2.2.3.

3.1 Messaging system for the SP-DevOps Observability process

As outlined in D4.1 *RC2: Scalable observability data transport and processing* measurements needs to be propagated from measurement functions up to control plane components located at more central positions in the network, in order to aggregate and process measurements before further communicate to even higher layers. Another related issue in RC2 is how and where to place aggregation points for processing, and efficiently filter data in order to keep the overall traffic overhead of measurement related data as low as possible. We identify a messaging system that could help OPs and MFs transfer data between them and make it available to the UNIFY architecture in an efficient manner.

3.1.1 Measurement data and control traffic transport

In order to communicate configuration, signalling, and measurement results between the monitoring and troubleshooting components (both measurement functions and aggregation points) within and between layers in the UNIFY architecture some kind of data transport protocol is needed. A naïve choice would be to use IP together with a transport protocol like UDP or TCP, and identify MF or OP instances using IP address together with the UDP/TCP port number. However in an environment like the one we envision in UNIFY this can be problematic for several reasons, 1) topological dependency of IP addresses, 2) components that are unsuitable for terminating IP, 3) coordinating IP address assignment over multiple domains, and 4) managing multiple TCP/UDP connections within the measurement components themselves.

The effect of 1) is a well-known problem where the address, which is bound to a certain topological location in the network, is also used to identify a service. If the service changes location in the network its identity has to change as well. In the UNIFY environment where services, as well as MFs, may migrate to different locations in order to handle failures, service degradation, or to deal with a changing demand this could become an issue requiring some way of maintaining a separate mapping of services to their current addresses, e.g. through a database service or placing services names in DNS.

MFs or OPs unsuitable for terminating IP traffic may for example be monitoring modules inserted into the software part of an OpenFlow switch or into other existing software or hardware. OPs that exist as part of an OpenFlow switch could be reachable through the existing OpenFlow

protocol but would require protocol extensions in order to be manipulated, or in other cases may require the invention of new protocols. This could be the correct approach in a longer term however for prototyping purposes it might be an unnecessary effort adding complexity to the prototype, and complicate prototype integration.

Finally, 3) and 4) complicate the development of the orchestration / control layers in WP3/WP5 as well as the development of individual monitoring components. A MF or OP whose function is not inherently complicated may become a large development task if multiple connections have to be managed and maintained during development, as well as a scalability or resource consumption bottleneck. For example there may be one or more connections open to send measurement data aggregates to higher layers, other connections to signal other components involved in the measurement, and yet other connections to e.g. a logging server in order to record certain events.

3.1.2 Message queuing systems

In order to handle these issues a messaging system can be deployed on top of existing connectivity in order to send messages between applications using protocols suitable for the individual applications connected to the messaging system. There are many potential systems and protocols that could be employed to serve this role, each supporting different application protocols, reliability, performance, programming languages, etc. The various messaging systems also support and are optimized for different kind of messaging patterns, for example a Request-Reply pattern between two applications, or a Publish-Subscribe pattern where one application generate messages that other applications selectively subscribe to. A quick overview of messaging systems [10] lists over 20 different; here we focus on some of the most popular ones. In *Table 1* six different messaging systems are shown, together with supported programming languages. We consider that, for prototyping work involving many different partners, supporting many languages is an important point, leaving ZeroMQ and RabbitMQ in the lead.

Language	ActiveMQ	Apollo	HornetQ	Qpid	RabbitMQ	ZeroMQ
C	Yes	-	-	Yes	Yes	Yes
C++	-	-	-	Yes	Yes	Yes
Erlang	-	-	-	-	Yes	Yes
Haskell	-	-	-	-	Yes	Yes
Java Message Service ¹	Yes	-	Yes	Yes	-	-
Java API (specific to a messaging system)	Yes	-	Yes	-	Yes	Yes
.NET	-	-	-	Yes	Yes	Yes
Objective-C	-	-	-	-	-	Yes
Perl	-	-	-	-	Yes	Yes

¹ Java Message Service is a standardized API than can be used to connect to various messaging platforms

PHP	-	-	-	-	Yes	Yes
Python	-	-	-	Yes	Yes	Yes
Ruby	-	-	-	Yes	Yes	Yes

Table 1: Supported programming languages for various messaging platforms

RabbitMQ [11] implements of the Advanced Message Queuing Protocol, and relies on applications connecting to a centralized broker using TCP/IP, meaning that all messages have to go through a centralized queue before being forwarded to applications. To scale the performance of this architecture RabbitMQ supports clustering which combines multiple instances into one logical broker, and federation which allows different brokers to connect and be located in e.g. different data centres. RabbitMQ supports many advanced reliability features such as persistent message queuing, storing messages in the broker in case of an application crashing, and high-availability queues through clustering to protect against e.g. hardware failure on the machine running the broker. RabbitMQ requires the clients to connect to a broker using TCP/IP, RabbitMQ supports many common messaging patterns like 1-to-1 messaging, load-balancing 1-to-N, 1-to-N publish/subscribe, 1-to-N selective publish/subscribe etc.

ZeroMQ [12] is a lightweight messaging system designed for high throughput and low latency scenarios, and supports both centralized broker architectures as well as broker-less ones. ZeroMQ lacks the message persistency and high-availability features available in RabbitMQ, unless one implements them on top of ZeroMQ. On the other hand, it is fairly easy to implement advanced messaging scenarios by combining the different components provided by ZeroMQ into a customized messaging system. Contrary to RabbitMQ, ZeroMQ supports several transport protocols to connect applications to applications, applications to brokers, and brokers to brokers. The supported protocols are:

- inproc, allows different threads in a multi-threaded program to exchange messages
- IPC, allows different processes on the same computer to exchange messages
- TCP/IP, allows messages to be exchanged over an IP network
- Transparent Inter-process Communication (TPIC), a protocol designed for intra-cluster communication
- Pragmatic General Multicast (PGM), a reliable multicast protocol running on IP

ZeroMQ also supports the same messaging patterns as RabbitMQ, however without the same level of reliability. The possibility to attach applications inter-thread, inter-processes, and inter-IP to the same messaging system gives a very flexible messaging solution well suited to the UNIFY monitoring architecture.

For rapid prototyping ZeroMQ seems to be a good choice. We also note that the jCatascopia cloud monitoring framework developed by the FP7 CELAR project that we reviewed in D4.1

employs ZeroMQ, so the choice of this framework may open avenues for inter-project collaboration at a later stage. If later during development needs for the reliability functionality provided by RabbitMQ would arise, steps can be taken to provide that. One solution could be to route messages that need that type of functionality to a RabbitMQ broker through a bridge between the systems.

3.1.3 Deploying a monitoring system using ZeroMQ

To more concretely show how ZeroMQ could provide a messaging service for monitoring control and measurement data, a generic model is presented in Figure 3.1. The clients in the figure can send and receive messages to/from each other using the service provided by the brokers. Clients and brokers may exist in various layers in the UNIFY architecture, connecting clients in many layers. DEALER and ROUTER represent different kinds of ZeroMQ socket types, one for 1-to-N communication, and the other for N-to-1 communication.

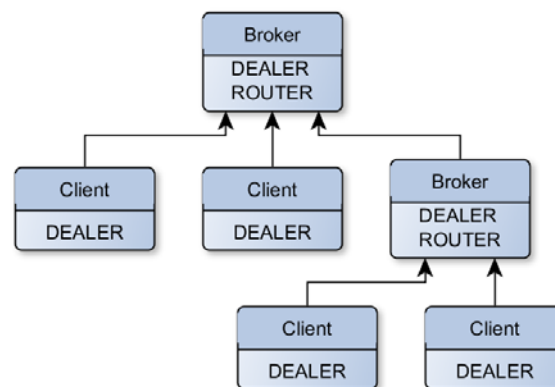


Figure 3.1: Clients connect to brokers using one of the supported protocols; brokers connect to each other in the same way.

In Figure 3.2 a more concrete example with OPs as monitoring components are shown. In each of the two Universal Nodes a broker is running, connected to a higher layer broker using TCP/IP. Monitoring components within the Universal Node connect to their broker using the IPC protocol, and can communicate with each other locally in a Universal Node or with components in another Universal Node via the higher layer broker. Other components such as “Monitoring CP3” and the “Logger” can connect directly to the higher layer broker and use it to communicate with all the monitoring components. This example only shows the 1-to-1 messaging pattern, a single message can only pass between two clients. For many-to-one, or one-to-many, PUB and SUB type ZeroMQ sockets can be added to the brokers and clients to support this kind of message flow.

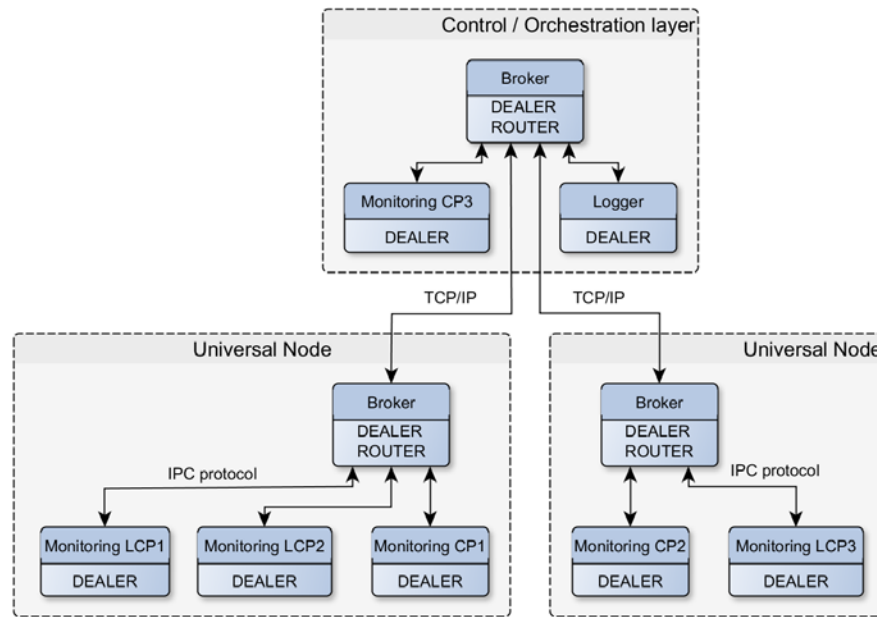


Figure 3.2: Using the messaging system to connect Monitoring LCPs, CPs, and a Logger. Some components are running inside Universal Nodes and others at higher layers, e.g. in the orchestration layer.

Returning to the SP-DevOps Research Challenge 2, the goal of reducing the overall network load can be targeted by relatively simple message routing in the brokers control and measurement data traffic can be kept as local as possible with regards to the overall network topology. The goal of optimizing the placement of Aggregation points (for example, “Monitoring CP3” in Figure 3.2) could likely be simplified by combining the knowledge of the physical network topology and the message bus topology. Assuming that the message bus topology is a tree, one optimal placement of an Aggregation Point could be e.g. at the node in the tree closest to the Monitoring LCPs (leaves in the tree) that it communicates with, since this represents the lowest number of hops for the measurement data to traverse. However, many considerations could come into the definition of “optimal” in this scenario.

An initial prototype of the client and broker has been implemented together with a simple message routing algorithm and been made available to the UNIFY project as a whole. While the prototype is fairly basic it illustrates how a ZeroMQ-based messaging system can be implemented and used for sending opaque data between applications. Implications on other work packages, primarily WP3 and WP5, are being discussed and will be reported in subsequent documents.

3.2 On describing monitoring functions

In software-defined infrastructure, we believe that the MFs and OPs need to be an integral part of programmability framework of the environment. Machine-readable descriptions of the capabilities and their configuration parameters need to be developed. They specify **which** measurement functions should be activated, **what** and **where** they should measure, how they should be **configured**, how the measurement results should be **aggregated**, and what the

reactions to the measurements should be. In collaboration with WP3, we are developing a “MEASurements, States, and REactions” language (MEASURE) that will be used to describe MFs and OPs.

So far MEASURE consists of three main components:

1. **Measurement definitions** which describe which measurement function should be activated, where the particular measurement should be taken, and how the measurement should be configured. These measurements can be existing measurement tools provided by hardware or software in the UNIFY architecture as well as tools developed by other partners in UNIFY,
2. **Zone definitions** which specify how measurement results should be aggregated and define thresholds for a combination of aggregated results, Zones definitions results in one or more finite state machines (FSM) whose movement between states depend on the aggregated results of measurements.
3. **Actions** which specify what actions should be taken both when moving between zones and while within a particular zone. Generally, actions consist of sending notifications, aggregated measurement results, and configuration to other components in the UNIFY architecture.

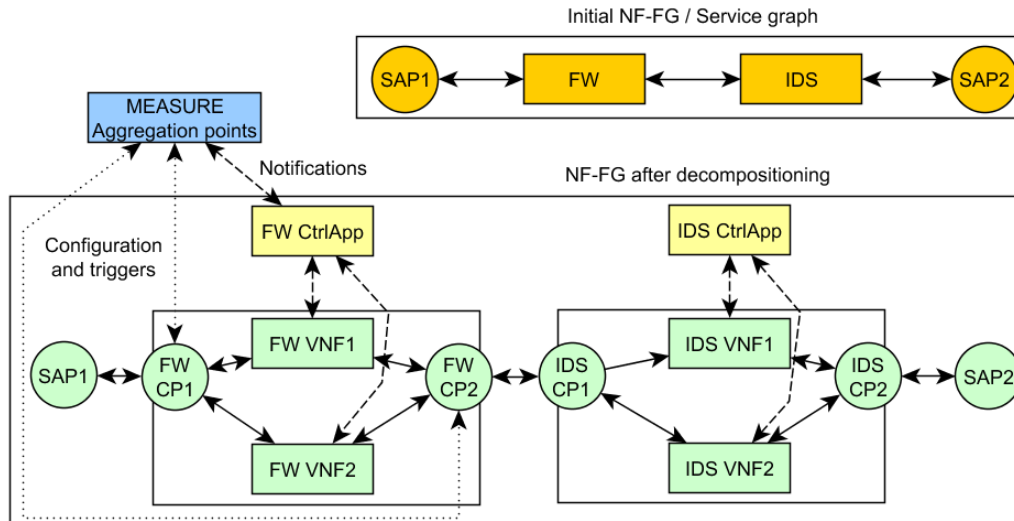


Figure 3.3: Configuration of delay monitoring of an instantiated Firewall VNF using MEASURE and the message bus to communicate configuration, notifications, and measurement results.

To illustrate how this could be used Figure 3.3 illustrates a use-case where a Service graph consisting of a Firewall and IDS system connecting two service attachment points have been instantiated in two Universal Nodes. The client or operator wishes to monitor the delay over the firewall and includes a MEASURE description in the Service Graph in order to specify what and how to monitor. The specification may look like:

Measurements: [$M1 = \text{Delay}(\text{FW-SAP1}, \text{FW-SAP2}, \text{param}=100\text{ms})$]

Zones: [$Z1 = \{ M1 < 50 \text{ ms} \}$, $Z2 = \{ M1 > 50 \text{ ms} \}$]

Actions: [$Z1: \text{Notify}(\text{FWCtrlApp}, 5\text{min}, M1)$, $Z2: \text{Notify}(\text{FWCtrlApp}, 5\text{min}, M1)$,

$Z1 \rightarrow Z2: \text{Notify}(\text{FWCtrlApp}, \text{"ERROR"}, M1)$, $Z2 \rightarrow Z1: \text{Notify}(\text{FWCtrlApp}, \text{"OK"}, M1)$]

The meaning of this snippet of MEASURE can be seen as the visualization of the aggregation point FSM in Figure 3.4 and interpreted as:

1. Initiate measurement function "Delay" between points FW-SAP1 and FW-SAP2, with delay function parameter set to 100ms (e.g. how often to perform the measurement).
2. Define two states, one when the measurement is below 50 ms and one when it is above or equal.
3. Define four actions: When in either of the state, send the measured value to "FW CtrlApp" every fifth minute. When going from one state to the other, send notification "ERROR" or "OK".

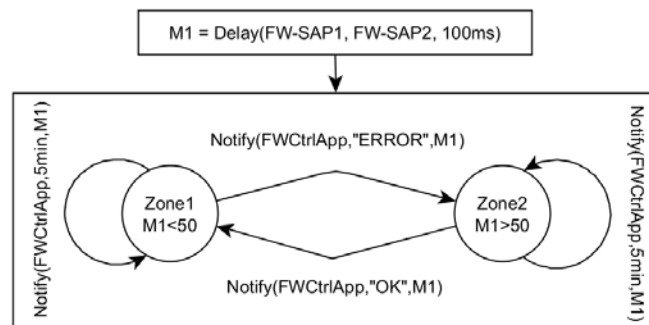


Figure 3.4: Example state machine defined by MEASURE. Based on incoming measurements the results the state machine switches between two zones and performs actions.

This simple example doesn't illustrate the potential aggregation arithmetic; zones could be defined based on average values over time, temporal parameters, the delta between two incoming values, etc. Actions could be used to affect routing decisions, modify measurement parameters, trigger additional measurements, etc.

An initial formal specification of the MEASURE Language exists but requires further adaption to the UNIFY architecture. An initial prototype of a basic aggregation point has been started. Discussions are ongoing on extending the language for including MFs and OPs developed part of the SP-DevOps.

3.3 In-network aggregation with VirtuCast

To perform fine granular and efficient monitoring of key performance indicators related to service chains, a scalable observability layer is required. A scalable transport of monitoring data therefore necessitates the usage of in-network aggregation. Virtualization of Observability Points

poses the question of where to place these points and how to efficiently steer the measurement information accordingly to the central aggregator.

Most works that consider the hierarchical deployment of distributed monitoring functions only allow for associative and commutative operations. However, many real world key performance indicators require for example the spatial or temporal correlation of data.

Only recently, algorithms emerged for trading off the number of observability points required, their placement in the network and the routing of measurement data towards these nodes, to balance bandwidth usage in the network and the computational overhead at monitoring nodes. However, as this is a new research field, only exact and optimal algorithms were considered until now. These algorithms are computationally expensive and therefore not directly applicable to the dynamic setting and scale in UNIFY.

The VirtuCast algorithm is currently under development and we published early results in [13] and [14]. These results show that finding near optimal aggregation overlays - including the placement of monitoring nodes - is feasible using Integer Programming approaches with execution times within minutes, while significantly improving over trivial solutions.

In *Figure 3.5* and *Figure 3.6*, the mapping of VirtuCast to OPs and NF-FG are shown. The diagram in *Figure 3.6* shows NF-FG after decomposition. Delay OP either connects to Delay Ctrl App or to Monitoring Function Control Plane (MF-CP). All DP Links are logical links or tunnels.

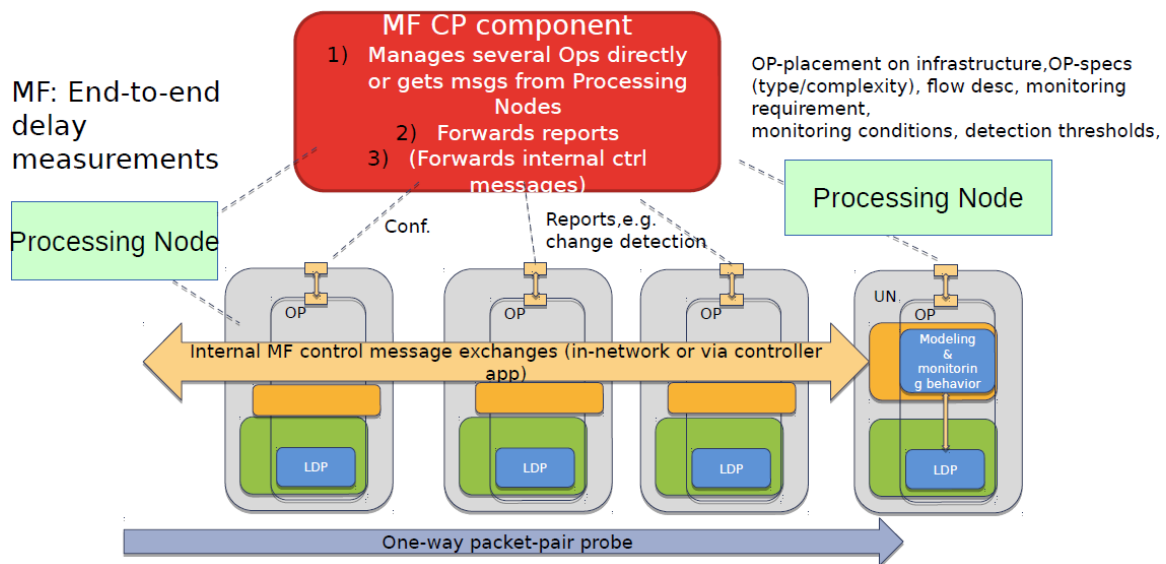


Figure 3.5: Mapping of VirtuCast to an MF.

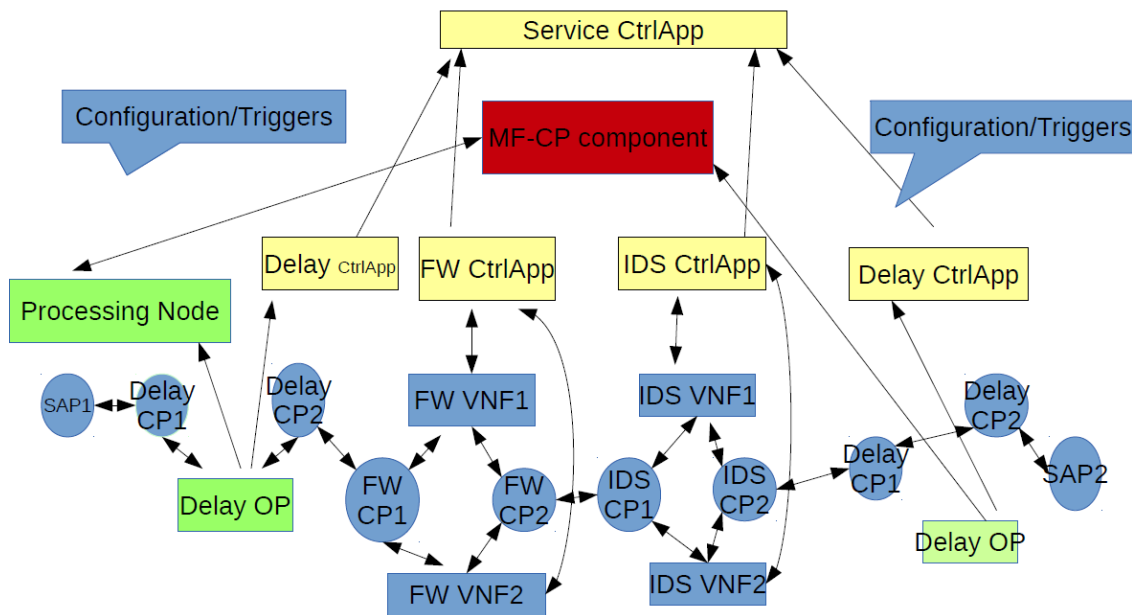


Figure 3.6: Mapping of VirtuCast to an NF-FG.

We have started considering heuristics based on Linear Programming and have obtained promising results. However, these heuristics still may take up to multiple minutes to re-compute solutions under changes in the deployment or monitoring objectives. Therefore, we will also experiment with purely combinatorial algorithms.

3.4 Configuration and update consistency

Updates to network policies and service chains are a reality: security policies may change, new demands arrives over time, or new traffic engineering strategies are pursued. However, updating networks and service chains is far from trivial: even in a logically centralized environment, it is hard to update networks consistently. The configuration of MFs and OPs is no different in this respect with that of any other VNFs. To a certain extent, it might be said that there are further constraints that apply to MFs and OPs because there are clear configuration dependencies on the VNF or virtual resource that is being monitored. We have started to investigate network updates which provide consistency guarantees. As a first building block, we consider the property of Waypoint Enforcement in a Software-Defined Network (SDN) environment.

3.4.1 State-of-the-art complement to D4.1

SDN enables a logically centralized and programmatic operation of computer networks. The envisioned network operating system has the potential to radically simplify the network management, as well as render the network more flexible: the software controllers can install, update and verify the paths that packets follow, i.e., the (network) policies, fast and in a globally consistent manner. However, today, we do not have a good understanding yet of the limitations of a more dynamic network management in general, and the SDN paradigm in particular. Over the last years, especially the problem of consistent network updates has received much attention. In a first line of works, initiated by Reitblatt et al. [15], network

updates providing strong consistency guarantees have been studied. They developed an algorithm that ensures Per-Packet Consistency (PPC) property even during the transition from an old policy to a new policy, i.e., each packet will either be forwarded according to the old (exclusively) or the new, but not a combination of both. In a second line of works, initiated by Mahajan and Wattenhofer [16], weaker transient consistency properties have been investigated for destination-based policies: during a network update, a packet may be forwarded according to the old policy at some switches and according to the new policy at other switches; however, the update still provides more basic transient guarantees, such as Loop-Freedom (LF): packets will never be forwarded along a loop. LF can be realized much more efficiently than PPC, and also does not require the tagging of packets.

We investigate fast updates for policies which describe arbitrary routes and are not destination-based: Indeed, the fact that routing decisions may not only depend on the destination, but also on the source or even the application, constitutes a key advantage of SDN, enabling interesting new opportunities for traffic engineering. Furthermore, arbitrary routes are also attractive because they reduce the dependencies between different paths to the same destination, facilitating even faster network updates. Moreover, we identify and initiate the study of a new fundamental transient property, namely Waypoint Enforcement (WPE). WPE is an important property in today's increasingly virtualized networks where functionality is introduced also in the network core. For example, in security-critical environments (e.g., in a financial institution), it is required that packets traverse certain checkpoints, for instance, an access control function (implemented by e.g., a middlebox, an SDN match-action switch, or a SP-DevOps MF implemented as a VNF), before entering a protected network domain. In other words, in order to prevent a bad packet from entering a protected domain, not only the old policy as well as the new policy must ensure WPE, but also any other transient configuration or policy combination that may arise during the network update. So far, waypoints could only be enforced using PPC, which by definition implies that new links can never be used earlier.

3.4.2 Contribution and First Insights

We initiate the study of network update problems where routing policies do not have to be destination-based but can describe arbitrary paths, and where weak transient consistency properties are ensured. First results are reported in [17]. In addition, we introduce an important new transient consistency property, namely Waypoint Enforcement (WPE), and show that at the heart of the WPE property lie a number of interesting fundamental problems. In particular, we show the following results: (1.) We demonstrate that WPE may easily be violated if no care is taken. Motivated by this observation, we present an algorithm WayUp [17] that provably updates policies in a consistent manner, while minimizing the number of controller interactions. (2.) We show that in contrast to other transient consistency properties, such as LF, WPE cannot always be implemented in a wait-free manner, in the sense that the controller must rely on an upper bound estimation for the maximal packet latency in the network. Moreover, the transient Waypoint Enforcement WPE property may conflict with Loop-Freedom LF, in the sense that both

properties may not be implementable simultaneously. (3.) We present an optimal policy update algorithm OptRounds, which provably requires the minimum number of communication rounds. The implementation of a first prototype is expected to take about 6 months.

3.5 Capabilities and tools for the SP-DevOps Observability process

The following subsections cover the progress of tools that were prioritized for supporting the SP-DevOps Observability process. Mapping to the NF-FG visual representation is part of the cross-workpackage integration activities. The visual representation using the NF-FG formalism reveals our understanding on how the lifecycle of resources associated to particular capabilities could be handled through the UNIFY Architecture. When the NF-FG specifications from D3.1 [4] will be transformed to a formal language description, we expect a rapid transition to the formal language and therefore a deep integration with the UNIFY resource management framework. The tools were outlined in D4.1 and address particular research challenges for operating software-defined infrastructure that were identified in the same document. Although we start with a strong focus on network resources, some of the work (for example, section 3.5.1) can be applied immediately to compute resources thus supporting the UNIFY combined cloud and network resource management framework.

3.5.1 In-network link monitoring

An important objective in the UNIFY project is the development of scalable monitoring approaches that in a resource-efficient manner can increase the overall network observability (RC1, D4.1). Data processing and monitoring tasks are usually carried out in dedicated network equipment, which impacts the scale at which monitoring can be efficiently performed as well as the overall observability of the network. Examples include simple SNMP [18] polling, ICMP [19] or more complex tools such as NetFlow [20] or sFlow [21] where sampled data and statistics are forwarded to a collection point for further analysis. However, dynamic service provisioning and performance management rely on the ability to perform scalable and resource-efficient measurements that can serve as input to richer statistical models of the observed network aspect, so that such information can be used in a proactive manner for resource management and quality assurance.

3.5.1.1 Link utilization monitoring

We base our approach on the simple and general idea of high rate but low complexity local updates and lower and adaptive rate analysis on locally produced statistics, which could then be distributed and collected as necessary. More specifically, link utilization monitoring is performed by recording statistics at high rate, which is used for updating parameter estimates at any lower rate suitable to the required precision of any given application. The statistical method that we propose is based on the use of two counters for storing the first and second statistical moments of each monitored metric, as compared to the current practice of using a single counter for each of the byte and packet rates. The use of two counters provides richer information about the observed traffic rates when used for parameter estimation of suitable distributions, which enables robust detection of performance degradations and resource-efficient dissemination of

the results (in terms of parameter estimates). Updating these simple statistics locally and at high rates allow for accurately capturing important aspects the traffic behaviour with significantly lower overhead than in a centralized setting. By adapting the rate at which the counters are queried, we can achieve flexible high quality monitoring without the cost of constant high rate sampling of the counters. Moreover, by inspecting the percentiles of cumulative density functions obtained from the parameter estimates our method can be used to detect increased risks of overload to a link in an extremely resource-efficient manner. Initial results (obtained from a stand-alone prototyping and simulation framework written in Scala) indicate that a log-normal distribution can be used to fit observed rates at a fairly high level of aggregation, and that episodes of high congestion risk at 0.3 s can be detected with over 98% accuracy using estimates captured at 5-minute intervals [22]. In the next steps we aim for developing a more robust detector that can be used for triggering for troubleshooting support and fault management, as well as for rescaling and resource re-optimization of NF-FGs in the infrastructure layer.

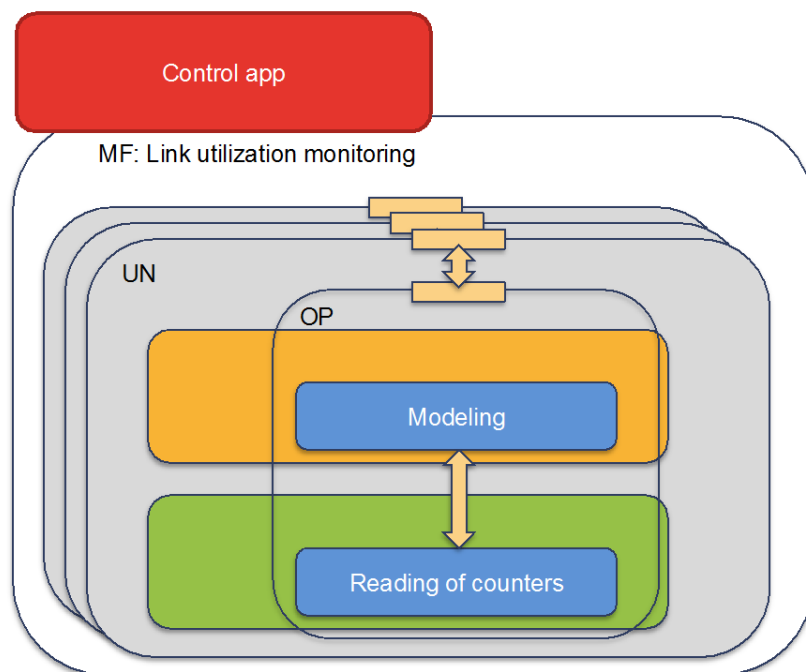


Figure 3.7: Example mapping of the link utilization monitoring function in the infrastructure layer

In *Figure 3.7* one example of how the MF for link utilization can be mapped to UNs is shown. The parameter estimates are performed locally in the node at the level of the VNF Management or VSE Management modules within the Unified Resource Manager, depending on the MF implementation, based on readings of the data plane counters storing the first and second statistical moments. The parameter estimates can be forwarded to higher layers in the architecture upon request or reported under specified conditions. An MF implementing link utilization monitoring can consist of one or several instances of link utilization OPs covering one or several parts of a mapped NF-FG, as illustrated in *Figure 3.8*.

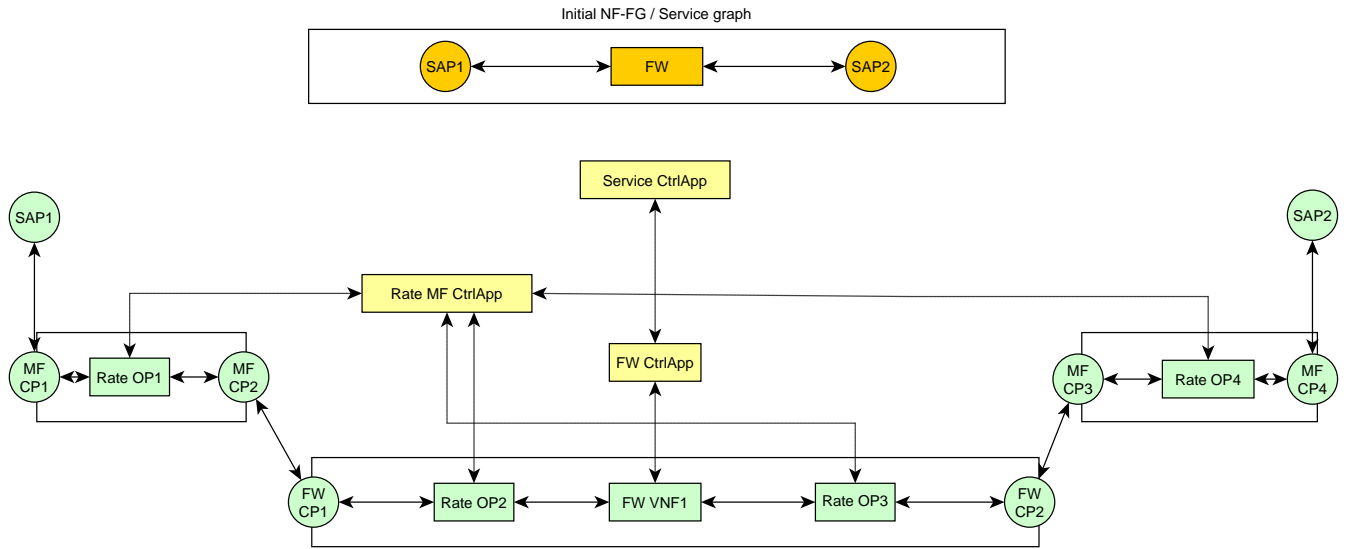


Figure 3.8: Example of mapped rate monitoring OPs for monitoring on Operator level

The mapping in Figure 3.8 shows a rate monitoring MF consisting of several OPs mapped on the infrastructure for monitoring on Operator level and controlled by a separate Control App.

3.5.1.2 Link delay and loss monitoring

Detecting performance degradations in a service-chain is crucial for dynamic service operation and quality assurance. However, measuring link delay and loss is generally difficult from an accuracy perspective as well as resource consuming, as it poorly reflects the service performance and often involve a large amount of probe and traffic injections. We address these issues by investigating a packet-pair approach where only the end-points of a monitored path are active, while flow rules in intermediate nodes are exploited essentially for packet forwarding and time-stamping. The measurements are obtained by observing one packet from the selected stream and then let another packet (injected, or selected from the stream for lower overhead) pick up timestamps along the monitored path for further processing. The obtained timestamps are used as input to a statistical model for deriving (via parameter estimation) the link delay and loss on intermediate links without explicitly measuring those link segments. The derived parameter estimates can then be used for detection and troubleshooting of changes based on comparing previous and current estimates as they evolve over time [23].

One example of how the method can be mapped to the infrastructure layer is shown in Figure 3.9, where the end-points consist of more complex OPs performing measurement control and modeling in the local orchestrator part of the UN, and the intermediate points consist of simpler OPs that mainly operate in the local Data Plane (DP) in terms of time-stamping. Note that the intermediate nodes can be UNs or simpler nodes with time-stamping capabilities and that the clocks for one-way delays need to be synchronized. The control app can manage OPs, forward reports to higher layers of the architecture as well as internal control messages between OPs in an MF and between other MFs. A mapped NF-FG can include one or several link-monitoring MFs

each consisting of a set of OPs as shown in Figure 3.7, for monitoring different parts of the NF-FG (Figure 3.10).

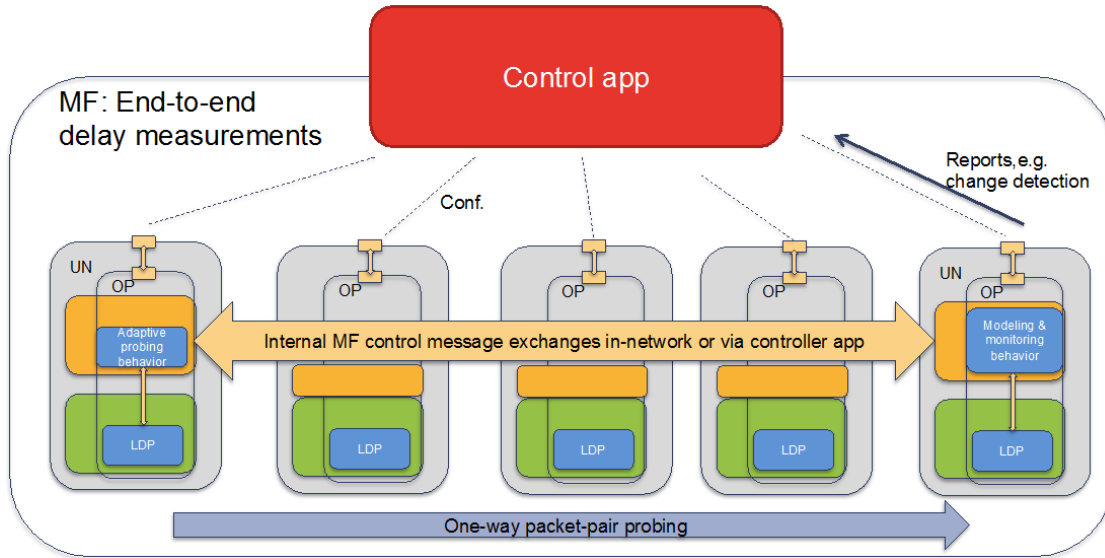


Figure 3.9: Example mapping of the link monitoring MF in the infrastructure layer.

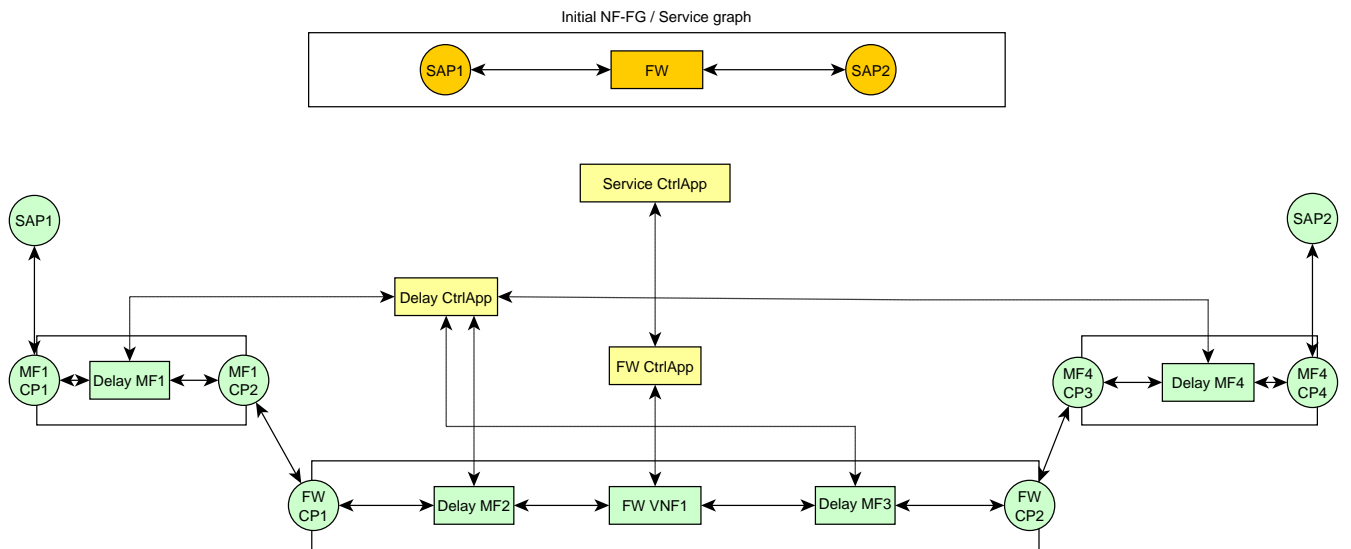


Figure 3.10: Mapping of several delay MFs controlled by one control app for the purpose of monitoring on Operator level. Each MF consists of a set of OPs, as shown in Figure 3.7

The approach contributes to the objectives of resource-efficient observability, as link metrics: 1) are obtained by using only two observability points performing more complex tasks (such as node-local modeling and probe selection); and 2) the intermediate nodes passively forwards packets and stores the timestamp of the latest match of a packet (e.g. as defined in [24]), which can be viewed as the simplest form of an OP. Moreover, the measurements reflect the link or service performance at any level of aggregation. Additionally, we are investigating statistical methods for implementing a self-adjusting monitoring behavior relative to specified precision

requirements on the parameter estimates. Such mechanisms allow for e.g. conditional observability and dynamic monitoring activity depending on the variability in the network, and contribute to reducing the influence (or overhead) of performing measurements. A NS3 simulator is currently under development for the purpose of method evaluation, and we expect to have initial results for the next deliverable D4.2.

3.5.2 Loss monitoring for aggregated flows

As outlined in D4.1, wide-scale usage of aggregate flow descriptors to describe network traffic in a SDN environment creates an opportunity for novel Operations, Administration and Maintenance (OAM) tools rely on user-generated packets and advanced functionality in the flow table for estimating packet loss and one-way delay measures. We report on representing the functionality as a WP4 Observability Point, mapping the functionality onto the NF-FG resource management formalism introduced in D3.1, as well as on the progress regarding the implementation.

The loss monitoring function that we outlined in D4.1 addresses research challenge *RC3: Low-overhead performance monitoring for SDN*. It is composed of two major parts: support within the SDN switch and application code executing on the controller. The representation as an Observability Point is shown in *Figure 3.11*, and the components will be explained in the following.

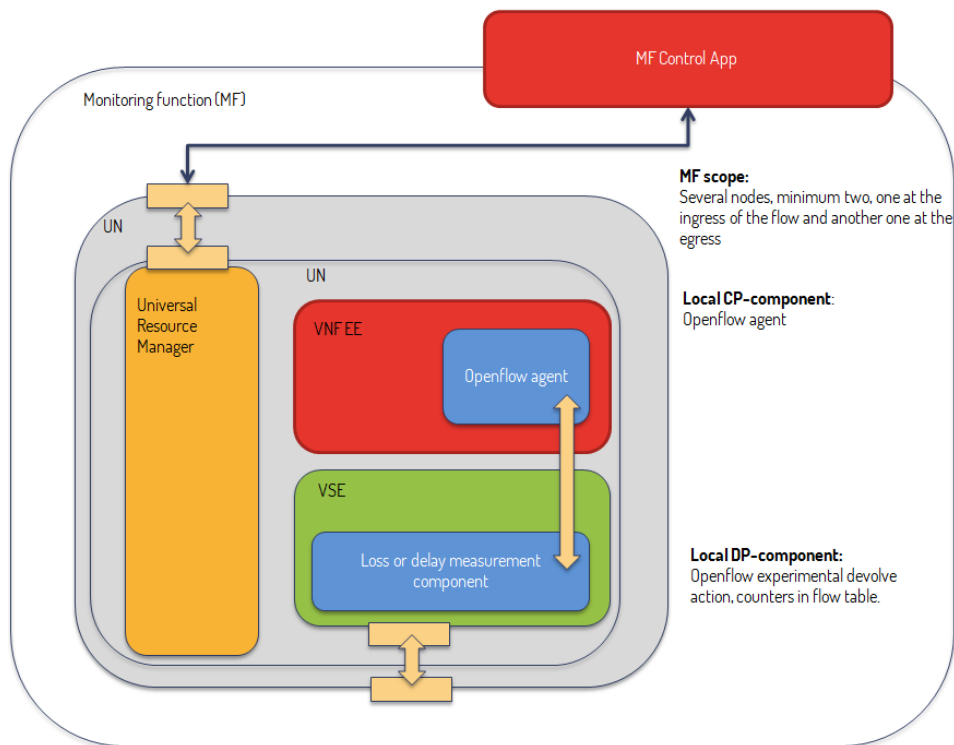


Figure 3.11: Example mapping of loss monitoring for aggregated flows tool as an OP

The support within the an OpenFlow-compliant Virtual Switch Environment (VSE) consists of enhancing the capabilities of the OpenFlow action processor in the VSE with a new action that implements the devolving operation as outlined in D4.1 e.g. creating a new rule in the flow table by taking information out of a packet that matched an aggregated flow descriptor. It also enhances the flow table with the capability to update a timestamp field for each packet that was forwarded through one of our devolved rules, in order to support one-way latency estimation in case the clocks of the ingress and egress nodes are synchronized. A local OpenFlow agent included within the VNF Execution Environment (VNF EE) (*Figure 3.11*) needs to be extended to support setting up the aggregated rule with the devolve action in the ingress node, and to setup the specific rule with timestamp action in the egress node. All these functions are represented within a generic “loss capability” in *Figure 3.12*. The ControlApp is the application that configures the measurement by choosing on which aggregate rules to enable the devolve action and policies for which individual flows should be chosen on the ingress node, as well as installing the appropriate flow rules on the egress node. The ControlApp also performs the calculation of the loss or delay estimate and makes the result available through a Northbound interface.

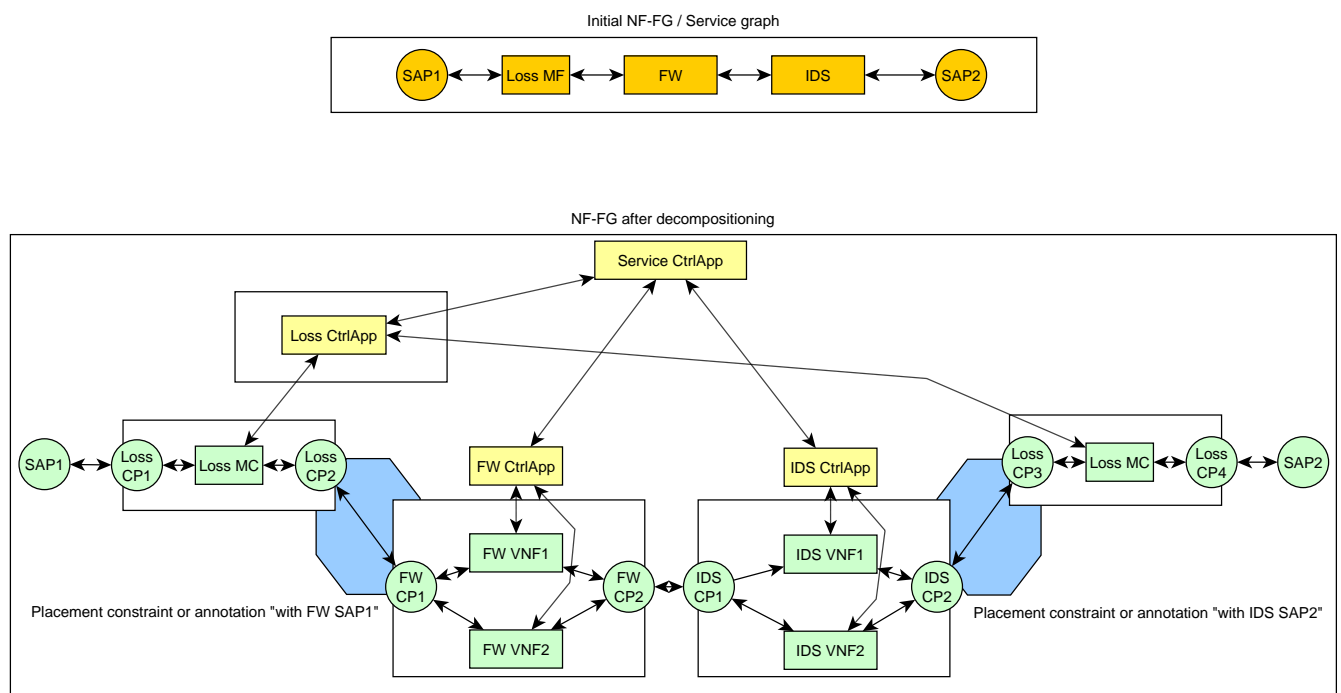


Figure 3.12: Example mapping of the loss monitoring for aggregated flows tool in the Service Graph

In *Figure 3.12*, we present a use case when our loss measurement function is included in a service graph associated with a scalable firewall (FW in *Figure 3.12*) and intrusion detection service (IDS in *Figure 3.12*). In this case, the Service Developer has chosen to instantiate a Loss Measurement Function in order to estimate the packet loss through the service. The Loss

Measurement Function needs to be available as a Virtual Network Function in the VNF catalogue of the system. During the decomposition process, the Loss Measurement VNF is expanded onto two Loss Measurement capabilities that are placed with restrictions (or using an annotation that is automatically generated) together with the Service Access Points at the ingress of the firewall VNF and at the egress of the IDS VNF. The ControlApp of the Loss Measurement is refined as part of the decomposition. Connections at the dataplane need to be setup between the ControlApp and the Loss Measurement Capabilities, as well as between the ControlApp of the Loss Measurement and the ControlApp of the overall service where the interaction with the Service Developer takes place.

In *Figure 3.13*, we show an option where the Loss Measurement Function is instantiated automatically based on policies setup by the Operator of the virtual infrastructure, for example in order to assist them with troubleshooting tasks. The Service Graph contains no representation of the Loss Measurement Function, and no data is transmitted towards the Control App of the firewall and IDS service. However, as result of the annotations introduced by the Operator, Loss Measurement Capabilities and their associated ControlApp are represented in the NF-FG and resources are allocated for them on the nodes of the UNIFY production environment. The only significant difference compared to the implementation option presented in *Figure 3.12* is the absence of the connection between the Loss ControlApp and the Service ControlApp, meaning that the results of the measurements will not be made available directly to a user of the service.

The implementation of the tool in an environment with the OpenDaylight controller, Mininet emulators and Open vSwitch software switches is ongoing. We expect to finalize it during the autumn and report initial evaluation results in the next deliverable.

4 Verification

This section details progress on defining capabilities related to research challenges we identified in the Verification area and documented in D4.1. As outlined in research challenges defined in Deliverable D4.1, verification activities are aimed at deploy-time functional verification of dynamic service graphs (research challenge *RC6 Deploy-time functional verification of dynamic Service Graphs*) and at run-time verification of forwarding configurations by enhanced ATPG (research challenge *RC7 Run-time verification of forwarding configurations*). These two kinds of verification are complementary in terms of properties verified, are executed at different times in the SP-DevOps Verification process, and are based on different verification techniques. Therefore, the tools progress is reported separately for each of them in the next sub-sections. Each verification technique could be integrated separately within the UNIFY Architecture. The two verification activities have a connection with WP3, and in particular the way the policies to be verified are received. The implementation of a common interface module for receiving these properties may be an idea for a possible preliminary integration step within WP4 and with WP3 in this area.

4.1 Deploy-time Functional Verification of Dynamic Service Graphs

Deploy-time verification is performed at different layers of the UNIFY architecture, during the various phases of deployment or re-deployment of service graphs. We report on an in-depth investigation to determine the potential for extending two established Openflow verification tools, VeriFlow [25] and Netplumber [26].

The verification activities at the highest layers (Service Layer and Resource Orchestration components) will be implemented by applying graph-theoretic algorithms to the Service Graphs (SG) and to the Network Function Forwarding Graphs (NF-FG), while the deploy-time verification activities at the Controller components will be implemented by adapting already existing specialized tools (that were reviewed from a theoretical perspective in [1]) that are capable of analysing the effects of the actual rules given to SDN Infrastructure controllers.

The final aim of verification is to improve the confidence in the fulfilment of certain properties related to SGs and NF-FGs (e.g., network node reachability, loop-freeness, black-hole presence, etc...), thus limiting the risk of critical and erroneous situations at deployment time. In order to achieve high confidence levels, we aim at leveraging formal methods, i.e. mathematically founded methods that can be used to prove that the involved models (SG, NF-FG, OpenFlow rules) fulfil certain properties.

In order to achieve at the same time high confidence levels and fast verification at deploy or re-deploy time, our idea is to split the overall formal verification task into sub-tasks, so that the sub-tasks to be performed at deploy and re-deploy time are simple (and hence fast) enough.

More time-demanding sub-tasks, instead, should be performed at VNF development time, and (once for all) when designing the overall verification methodology.

More precisely, at VNF development time, or when a new VNF is added, a formal model of the VNF must be developed and the fact that the VNF code correctly implements this model should be verified at that time. At deploy or re-deploy time, instead, verification is performed using the formal models of the VNFs (which are a representation and thus a simplification of the actual VNF software itself), and assuming that the VNF code correctly implements the model. Another assumption that will be made when performing verification at deploy time at the Controller component is that the interpretation of network-specific SDN rules by the real controller is consistent with the interpretation model used by the verification engine.

In UNIFY, we have decided to focus attention only on the verification sub-task to be performed at deploy and re-deploy time, while we do not address the verification of VNF code, for which standard software verification techniques could be used as already mentioned in [1].

Among the verification activities to be performed at the different layers, the most challenging one is undoubtedly the verification at the Infrastructure Controller Layer, because the tools that are currently available for this kind of verification are targeted to pure OpenFlow systems and their extension to cope with UNIFY-like systems is far from trivial, especially if we consider the aim of not decreasing their time performance and scalability features. For this reason, in the initial project phases, attention has been focussed mainly on this part.

A first important step towards the definition of an ad-hoc mechanism to verify network and security properties at the Controller Layer in the context of the UNIFY architecture has been the analysis and evaluation of the most promising tools in this category, i.e. the ones that can fit the UNIFY requirements, with a special attention to performance.

Considering the approach and the scalability achieved, the tools that most closely seem to meet our requirements are VeriFlow [25] and Netplumber [26]. These tools are based on different verification techniques but they share some important features:

- Both of them are designed to work in cooperation with a SDN controller, and use a view of the network-wide state obtained through the SDN controller, by analysing the OF rules issued by the controller, and blocking the unsafe ones or issuing alerts when some properties would be violated.
- Both of them are reported to run a verification taking a time in the range of some milliseconds on average, and seconds as a maximum, depending on the network complexity and on the kind of verification performed.

The main aims of our analysis and evaluation of these tools were:

1. Check the availability of the source code of the tools and its documentation.

2. Check that the performance results presented in literature are reproducible.
3. Compare the performance and scalability of the tools
4. Identify possible ways for extending the tools for performing verification in the UNIFY architecture.

The next subsections present the preliminary results of this analysis and evaluation work for the two selected tools.

4.1.1 NetPlumber preliminary analysis and evaluation

NetPlumber is a run-time policy checker for network invariants and reachability constraints on wide scale topologies. It exploits the Header Space Analysis [27] (HSA) as theoretical foundation. The HSA interprets each packet as a point into the $\{0,1\}^L$ space, where L is the length of the packet header. This geometric interpretation includes a model of each forwarding node as a transfer function that maps an incoming packet into a set of output packets being sent over one or more interfaces. NetPlumber is built on top of this framework and it is constituted by an agent that sits in line with the control plane and intercepts each status update from the OpenFlow controller to ensure that no property violation occurs. If it is the case, NetPlumber can discard the update, by removing the rule inserted in the involved nodes, and can also issue an alert for the network administrator, depending on the provided configuration. At each new relevant event, NetPlumber updates its internal representation of the network, called *plumbing graph*, and checks the property violation with respect to the network portion that is affected by the occurred event. As a consequence of this optimization, the tool is fast enough to largely satisfy most of the usual requirements in terms of verification speed.

The HSA library implementation along with the NetPlumber source code is publicly available in [28]. It consists of different tools and modules:

- a Python based implementation of the HSA library consisting of about 11k lines of Python code considering the examples provided and the network scenarios analyzed in the paper (not supported anymore);
- different types of router configuration parser, which are used to statically generate the transfer function modeling the behavior of some commercial routers (Cisco, Juniper, etc...);
- an efficient C based implementation of the HSA library, consisting of about 3k lines of C code, called Hassel Library;
- the C++ NetPlumber tool built on top of the C library, which consists of around 9k lines of code (including the library used to parse JSON and the library implementing the HSA main functionality).

Unfortunately, code documentation is not provided, hence any possible extension of this tool requires a complete analysis of the whole source code to understand all the implementation details.

An example of the entire verification workflow exploiting the above mentioned tools is sketched in *Figure 4.1*. The routers configuration files must be provided as input to a vendor specific parser that generates, as output, the corresponding transfer functions, in a generic format, and the network topology. All this information can be directly used by the Hassel library (or NetPlumber) to run reachability checks on a ports pair.

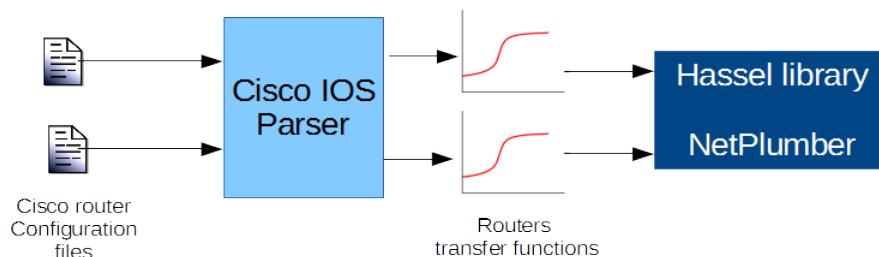


Figure 4.1: NetPlumber Workflow

NetPlumber has been validated by the authors considering 3 large use cases: (i) the Google WAN, a large SDN network deployment connecting Google data centers, (ii) the Stanford Backbone, a wide campus network comprising five /16 networks and (iii) Internet2, a national scale network supporting a huge number of US institutions. The performance reported in all the three cases are satisfying, ranging from some microseconds to, at maximum, few seconds for each status update verification with an average value in the order of some milliseconds.

In order to reproduce and validate part of the evaluation results reported in the original paper, a working environment has been setup to check the tool performance. In particular, the Stanford topology included with the tool has been considered with a customized script that selects random pairs of router ports in the given topology and runs a reachability test to identify traffic classes that could flow from the source port to the destination port. This test is significant since, as demonstrated in the paper, most of the security policies can be translated into equivalent reachability constraints. The aim of this test is collecting statistics about the performance achieved for each verification process. To this end, the script averaged the results from 10000 runs and the results are depicted in *Figure 4.2*.

Figure 4.2: NetPlumber performance in our testbed

Each point of the x-axis represents a test, conducted through the random selection of a pair of ports and executing the reachability check between them. On the y-axis the execution time for each test is reported, along with the average value (in blue) calculated over the red curve. The obtained results are consistent with the ones provided in the paper, especially considering the different hardware architecture used to run the test (A workstation characterized by 8 GB RAM, Intel i7-3770 @ 3.40 GHz CPU, Ivy bridge architecture with four cores plus hyperthreading running Ubuntu 14.04 LTS).

4.1.2 VeriFlow preliminary analysis and evaluation

The basic idea of VeriFlow is to analyze the rules effect just on those parts of the network that could be affected by the changes. Hence, the tool performs the following three steps:

- Slice the network into Equivalence Classes;
- Generate a Forwarding Graph for every equivalence class;
- Run pre-defined or custom queries on the graph to check the invariants.

An Equivalence Class (EC) is defined as “a set P of packets such that for any $p_1, p_2 \in P$ and any network device R , the forwarding action is identical for p_1 and p_2 at R ”. To compute such classes, VeriFlow stores network rules in a multidimensional trie structure that associates the packets matching a forwarding rule with the rule itself. In particular, this data structure collects all the installed rules in the network: for each rule, a node with three branches represents a single rule’s bit (which corresponds to a specific packet header bit), whose value could be 1, 0, don’t care, depending on the real rule bit’s value.

When a new rule is installed, VeriFlow looks up in the multi-dimensional tree to establish which installed rule intersects the new one. In this way the set of packets affected by the new rule is discovered and then the ECs are computed (*Figure 4.3*).

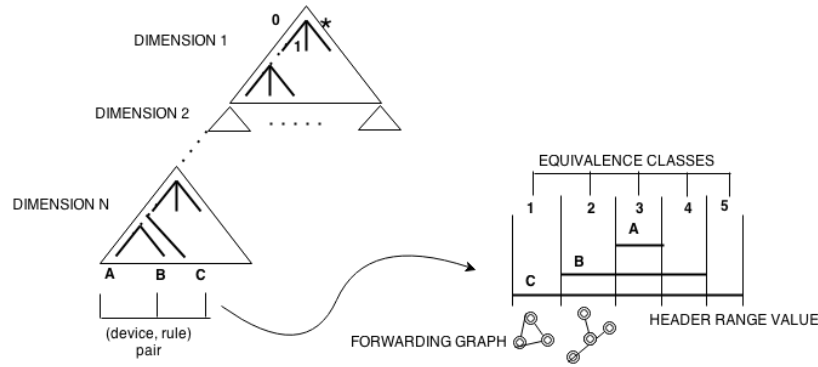


Figure 4.3: VeriFlow Equivalence Classes generation process.

After generating the ECs, for each of them a forwarding graph is computed to know how the packets belonging to an EC traverse the network. As mentioned before, VeriFlow verifies only the part of network that could be affected by the changes; hence, this tool generates the forwarding graphs corresponding to the ECs whose behavior can change after the rules modification.

The last step is the verification of a list of pre-defined network-wide invariants. An invariant can be seen as a verification function that takes as input the forwarding graph specific for one of the affected ECs by the new rule: on this forwarding graph, a set of queries is run to verify the invariants. As Figure 4.1 shows, if the new rule is acceptable, i.e. it does not violate any invariant, then the rule is installed in the network. Otherwise, an alert for the network administrator is issued and some possible actions could be suggested.

VeriFlow is written in C++. The source code of this tool is made available by the authors on the web-site [29]. VeriFlow is composed of about 5k lines of code. To summarize the architecture of the code, Figure 4.4 shows the UML diagram of all the VeriFlow classes. The most relevant classes are the ones that implement the concepts described so far (e.g., VeriFlow, Trie, ForwardingGraph, etc.).

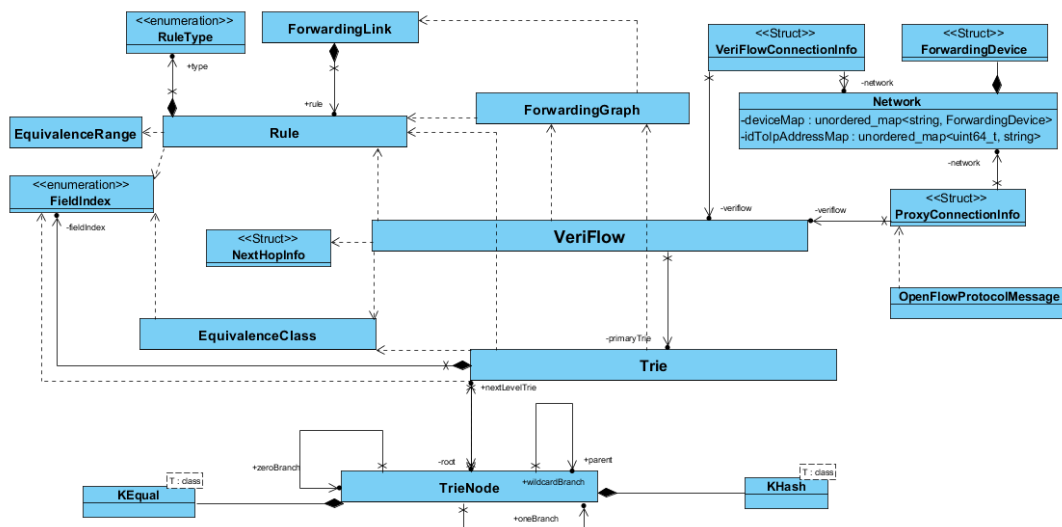


Figure 4.4: The VeriFlow UML class diagram.

Even if VeriFlow has an easy to understand approach, it seems that no documentation is available except the two papers, hence any possible extension of this tool requires a complete analysis of the whole source code to understand all the implementation details.

As it is described in [25] [29], VeriFlow is implemented in two modalities:

- Integrated in the NOX controller;
- As proxy layer between a SDN controller and the network.

While in the first version the author expects an integration with other SDN controllers like Beacon, Maestro and Floodlight, the proxy mode seems not to have any constraint on the specific controller used.

An interesting feature of this tool is that it allows defining a custom topology in a simple format. For example, a switch is characterized by a set of parameters like an identifier, an IP address and a list of port-address pairs to define how the switch itself is connected to the rest of the network. VeriFlow exposes an API to write new invariants, among them some have been already defined like node reachability, black holes and loops: both new and pre-defined invariants can be tested on custom topologies. Another useful characteristic of VeriFlow is that it is implemented supporting the OpenFlow protocol and also the IP forwarding rules.

To test VeriFlow's effects on TCP connection latency, we emulated an OpenFlow network using Mininet [30] and made the network topology description available in the distribution package. This network was composed of 20 nodes: 10 switches and 10 hosts that act as TCP client and server. In particular, this network is controlled by the NOX OpenFlow controller by means of an application that acts as learning switch.

Basically, VeriFlow should add an overhead due to both the verification module and the fact that it acts as proxy layer between the network and NOX. We measured this overhead in three scenarios:

- Without VeriFlow;
- When VeriFlow acts as a simple proxy and no invariants are verified;
- When VeriFlow acts as proxy between NOX and the network and the verification module works.

Figure 4.5 shows how increasing the number of hosts that set up TCP connections with each other, the setup latency of the TCP connections grows. Of course when VeriFlow is active, the TCP latency is higher, but the relevant thing is that there is no evident difference between when the verification module is active and when it is not.

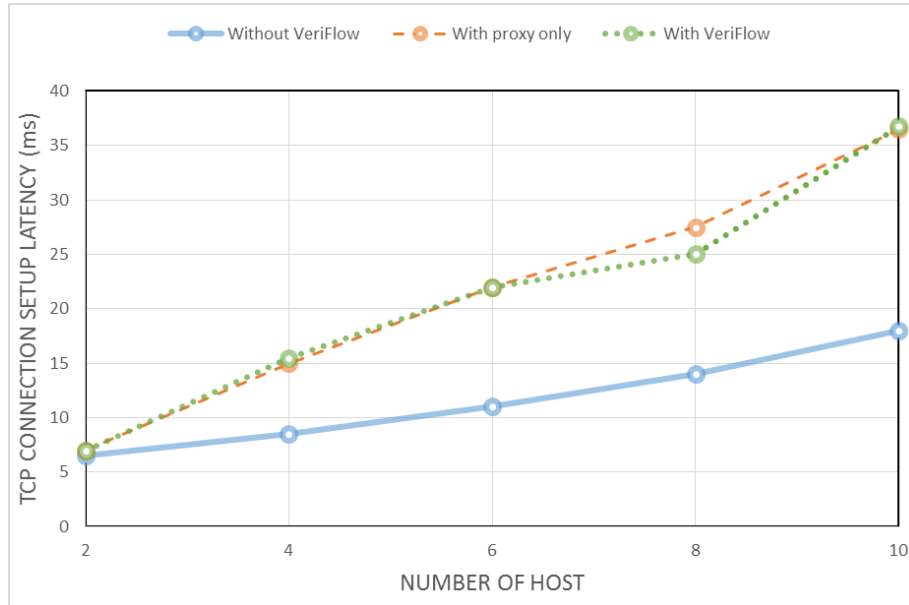


Figure 4.5: VeriFlow TCP connection setup latency.

As this test has been performed on a relatively simple network, it will be important to test how VeriFlow behaves with more complex use cases, in order to better assess its performance and scalability in comparison with the ones that can be obtained by NetPlumber.

4.1.3 Conclusions and planned future steps

One of the main limitations of both the approaches of NetPlumber and VeriFlow is the inability to model boxes with an associated internal state (for example a NAT or a stateful firewall). In fact, the HSA is based on the construction of transfer functions that only depend on the static configuration of routers and process packets just by looking at their header (or part of it). Similarly, the approach of VeriFlow based on equivalence classes does not take into account that the state of some network functions can produce different forwarding decisions into the network: VeriFlow just considers stateless functions that receive packets and modify them only on the base of the current input.

This limitation could be overcome by taking periodic snapshots of the network and running again the verification in order to capture possible updates of the network state and continuously ensuring its correctness. Clearly, the scalability of this solution could suffer from an excessive updates frequency (even with the re-computation of equivalence classes in the case of VeriFlow), thus limiting its range of applicability.

Both the considered approaches intercept the changes to the network driven by the OpenFlow controller and update their internal representations of the network. This is possible because the controller is the central point in the network where routing decisions take place, hence checking network properties violations is allowed at run-time, in a dynamic manner. However, the OpenFlow scenario is conceptually different or at least less general than the one envisioned in the context of UNIFY for at least two reasons. First, the presence of the active functions within

the network should imply a more general model to represent all the possible applications that participate in the UNIFY vision, even the ones making use of state information. Second, in UNIFY there could be multiple points in the network where routing decisions can be made (in general, each active function can modify its routing table dynamically) hence there is no single node having a complete view of the network (like the controller in the OpenFlow case).

Some work in the direction of overcoming the first two limitations was recently made available by a team at UC Berkeley who submitted a draft preprint on the topic to arXiv [31] late September 2014. This first attempt addresses isolation properties and shows a way for verifying these properties in the presence of stateful middleboxes. According to the results presented in the paper, this verification technique seems compatible with our performance target. As it is the first paper that addresses the problem, there are margins for extension and improvement.

A third limitation of both VeriFlow and NetPlumber considered here is that an incremental network configuration change cannot be properly verified: for example, a SDN application that has to be installed in the network could be translated into more than one rule, and these rules are sequentially deployed into the network. Hence, working as a proxy between the network and the control plane, these approaches are not able to recognize if an invariant violation is due to an intermediate (i.e. transient) network configuration state and can be admitted only for a small time interval, or if it is a real property violation.

Our plan is to overcome the abovementioned issues by extending the range of applicability of NetPlumber or VeriFlow to the UNIFY use cases. To do so, the next planned steps are:

1. Complete the experimental evaluation of VeriFlow and NetPlumber, by using them with the same use cases, thus being able to fairly compare their performance and scalability.
2. Based on the results of evaluation, define possible ways for extending the approaches of VeriFlow and/or NetPlumber to active network functions and non-pure OF environments. A central point of this activity will be defining how to model stateful VNFs. The work will take into account the approach recently developed at UC Berkeley, trying to exploit it and possibly go beyond it.
3. Check the defined extensions on use cases.

4.2 Run-time Verification of Forwarding Configurations with AutoTPG

For the run time verification of network configurations and services, network engineers have to debug errors related to configurations and performance issues such as forwarding rules errors, forwarding actions errors, link failures, bandwidth usage, latency, and loss. To debug these errors, the Automatic Test Packet Generation (ATPG) tool [32] was already proposed in literature. It gathers the forwarding table information and uses it to calculate the minimum number of test packets to transmit in the network in order to validate the expected configuration. Gathering the whole forwarding table information causes scalability problems when using ATPG since the amount of gathered information will grow with the network size.

From a verification perspective, ATPG is only able to verify and test the action part of the Openflow forwarding rule, leaving the matching part of the forwarding rule untested. Furthermore, only active rules can be tested by ATPG, for example in a case with both an active and a backup rule for failover, an error in the backup rule cannot be detected during normal operation. Vice versa, an error in the regular path cannot be detected while the temporary backup path is active during a failure event.

The main goal of our enhanced test packet generation tool, AutoTPG, is to solve the aforementioned limitations, firstly by reading the counters of ports or some of flows instead of reading the complete forwarding tables, secondly by testing both the matching- and action part of a rule, and finally include inactive rules for protection paths in tests. We expect that this methodology will decrease the information gathering overhead and will decrease the scalability issues of ATPG. In addition, as we do not want to apply our testing algorithm on data packets traffic, we define an EtherType for test packets (similar to the OAM EtherType of IEEE 802.1AG) to distinguish these packets from data packets. AutoTPG can be used to verify the system during the development, deployment, and operation phases of the SP-DevOps cycle. Using AutoTPG, an SP-DevOps Operator or VNF Developer can verify an Openflow forwarding path for link failures, header matching failures, and the existence of forwarding loops.

To test the link failures we add a forwarding entry in an OpenFlow switch through which virtual machines configured in the network transmits test packets from all the ports of the switch and the infrastructure controller then periodically reads the counters of the receiving ports of the neighboring switches. If there is no increase in the packet counters of a specific port, we declare that there is a link failure between the switch transmitting test packets and that port. This procedure significantly decreases the ATPG scalability problems as information related to only port counters is gathered. In addition, the numbers of packets transmitted in the network is significantly lower than the number calculated by the ATPG algorithm, as the packets are transmitted only to the neighboring switches and the neighboring switches just drop these packets.

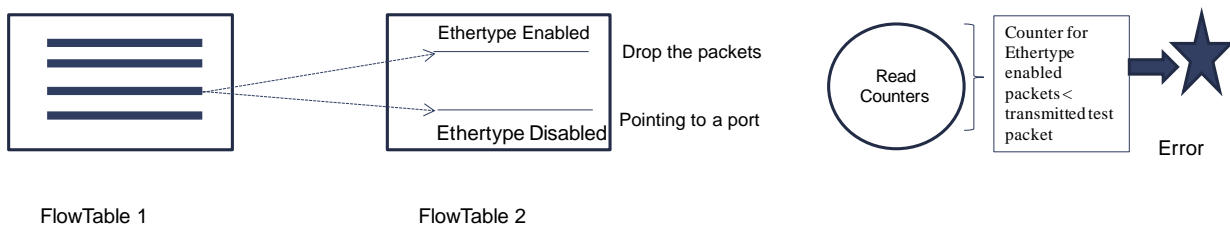


Figure 4.6: Matching header failure detection

For detecting matching header failures, our testing algorithm first detects the flow entries installed by the controller and then tests these flow entries for the matching header failures. For detecting matching header failures, we test each kind of packet that may traverse an

Openflow flow table entry, with or without wildcards. We add three entries to an OpenFlow switch flow tables to test a flow that was already defined (*Figure 4.6*), one entry in Flow Table 1 and two entries in Flow Table 2. The first entry has the same matching header as the flow and has the wildcarded EtherType. This forwarding entry redirects the packets to the second forwarding table in which there are two entries containing enabled or disabled EtherType. The enabled EtherType entry will drop all the packets (test packets) and the disabled EtherType entry will forward the packets (data packets) to the right port. For the matching header failures, we read counters of the forwarding entries of the second forwarding table to find which type of packets give errors in the matching header field. To find the errors in a flow entry for different types of packets, packets in question will be sent to the switches with the enabled EtherType and an error is generated if the counter of enabled EtherType forwarding entries is less than the transmitted packet. To decrease the number of counter updates, virtual machines first send all the types of packets that are needed to be tested. If the counter of these entries is less than the number of transmitted packets, we assume errors in the defined forwarding entries. To find the exact errors in the types of packets, virtual machines now send the half of the packet, and find the errors using the same algorithm described above and if there are no errors found using this number of packets then virtual machines transmit the other half and find the errors. We always transmit the half the number of packets until we reach to point in which all the types of packets which cannot be handled by a flow entry is calculated.

For forwarding loops in the data plane of the infrastructure or within a NF-FG, we first detect potential loops by analyzing forwarding table information stored in the controller. Nodes suspected to have loops then are tested by sending packets in the network. To perform this test we automatically create virtual machines in the network and copies of transmitted test packets marked with sequence numbers are transmitted to the targeted virtual machines. If multiple packets with the same sequence numbers are received, a suspected forwarding loop has been verified.

One example of how the AutoTPG tool can be mapped as an OP to the infrastructure layer and to the virtualization layer is shown in *Figure 4.7*. In this example the end-points are test packet generation and analytic components implemented as virtual machines, these are configured by the Control App to transmit, receive, and analyze test packets.

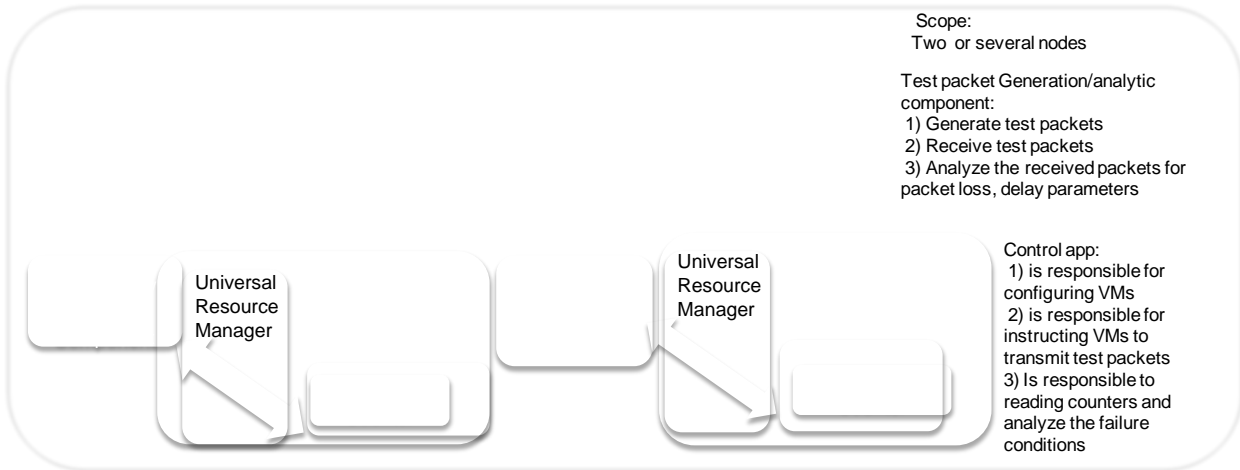


Figure 4.7: Example mapping of the AutoTPG tool in the infrastructure layer/virtualization layer

AutoTPG not only contributes to the SP-DevOps Verification process, it can also contribute to the SP-DevOps Observability process by enabling packet loss, delay, and throughput testing. For packet loss and delay estimation, AutoTPG transmits test packets from one end point of in the network to another and calculates the number of lost packets and measures the transmission delay which occurred in the network. If the delay is more than the expected or defined by an SLA, AutoTPG can send a report to higher layers.

In addition to link failures, matching header failures, forwarding loops, we also plan to define a capability to measure the throughput of a link (bandwidth usage and latency) using an algorithm similar to traffic generator algorithms such as iperf and DITG [33].

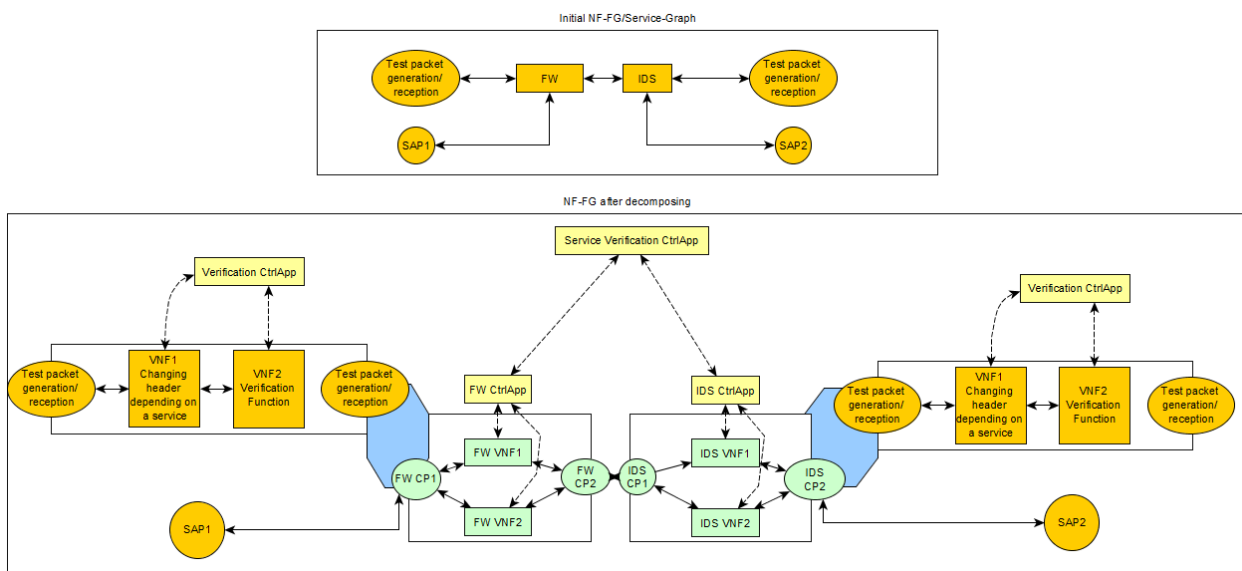


Figure 4.8: An NF-FG mapping for the AutoTPG tool

In Figure 4.8, we represent the AutoTPG as a network function requesting resources through an NF-FG mapping. In this scenario, test packets generated by a test agent go through a virtual

network function (VNF) which modifies the test packet header to match the expected header in the service. The modified test packets then go through another VNF, which performs the troubleshooting functions (e.g., counting the number of packets), finally the test packets are transmitted through the network. The packets will then reach through the service access point (SAP) Firewall and the root cause of the faults such as link failures, packet loss etc. will be determined by the verification controller app attached to the IDS system shown in the figure. At the end the test packets reach the AutoTPG test packet logic which reverses the test packet header modification and performs the troubleshooting functions.

5 VNF Development Support

D4.1 introduced research challenge RC11 *VNF development support* that highlighted different aspects of the Service Provider DevOps from the perspective of VNF Development Support. Since then, we identified three main underlying research challenges that should solve the problems of network isolation, visibility, and flexibility.

Network isolation means that traffic of a NF-FG under test should be isolated from the rest of the network. Obviously, isolation in case of normal operation is required as well; however separating traffic of different services is not trivial as it might interfere with advanced traffic steering methods. For example, none of the two traffic steering methods presented in [34] protects against misbehaving or faulty network functions, because in these cases the routing decisions are based on information that untrusted VNFs add to data packets.

By visibility we mean that the components of the UNIFY Architecture should provide accessible debugging interfaces for its functional blocks. Since the architecture is inherently built on indirections, a user needs to traverse through these indirections to access a specific functional blocks.

Finally, the UNIFY Architecture is still taking shape, hence a tool addressing VNF Developer needs should be flexible and prepared to accommodate changes in the architecture. Moreover, it should provide debugging capabilities to existing components like, for example, an Open vSwitch instance or an OpenDaylight controller instance.

To address these challenges, we build a multilayer, multicomponent debugging tool. The next section presents the current status of the work. Moreover, we also created a stand-alone debugging tool providing *network watchpoints* that provides visibility onto the OpenFlow control channel and two actions that facilitate troubleshooting activities. This tool (presented in Section 5.2) could be used stand alone, but it also served as a case-study for the multicomponent debugging tool and will be integrated into the multicomponent tool.

5.1 Multilayer, multicomponent debugging tool

When it comes to troubleshooting or debugging distributed, multi-component architecture there are two main approaches. First, a workflow-based approach as used in Service Oriented Architectures treats components as black boxes and focuses on the interactions of the components. By examining the information passed between components the developer can find deadlocks or livelocks. Second, in the component decomposition approach the developer examines the behaviour of each component in isolation. We have chosen to follow both of these approaches in parallel.

For the sake of flexibility, we plan to develop a universal interface that helps integrate different components to the debugging tool. Using this interface, a component should be able to describe its capabilities and the VNF debugging tool should be able to automatically connect these

components. For example, an IP address in the topology description could be used to query the state of a VNF running on the host or resource using that IP address.

Although the universal debugging interface is at an early stage, we have created proof-of-concept interfaces to Open vSwitch and three well-known SDN controller frameworks (Floodlight, OpenDaylight and POX). We relied on the popular Emacs text editor from a Linux operating system to provide a common user interface for accessing these OpenFlow controller frameworks. Although the industry seems to prefer an Eclipse integrated development environment over Emacs, it is more natural to extend Emacs with the required functionality than Eclipse. One of the strong sides of Emacs is its superior support of interacting with sub-processes. For example, establishing a shell connection to the host running a VNF instance is arguably easier in case of Emacs. Moreover, shell buffers are integral parts of Emacs, whereas they are only available in Eclipse as add-ons requiring further efforts for integration. We developed Emacs sub-processes to display port statistics and flow tables of Open vSwitches. Also, using the integrated so-called “gdb-mode”, we can instruct Emacs to attach a Linux software debugger to VNF instance running on a remote host.

The implementation activities had two focuses. First, we investigated how different external application could connect to Emacs for providing a troubleshooting or debugging user interface to VNF Developers. Second, we connected these user interfaces in different ways. For example, the network topology information is visualized using the COGRE module of Emacs and we added links to the switches in COGRE buffer through which flow tables of switches can be queried. This simplifies the work of a developer as she no longer needs to open a shell connection manually, log-in to a remote host, and execute the “ovs-vsctl” command from the command-line interface.

5.2 Network watchpoints

We developed a standalone debugging tool that helps to define *network watchpoints* for certain datapath or control traffic events (policy violation, or any OpenFlow matching rule) that trigger different actions. Although the network watchpoint is fundamentally similar to software watchpoints or breakpoints, it is not capable of pausing the operation of the physical or virtual network switch when it is triggered. To suspend temporarily the packet processing in SDN devices, a massive and comprehensive hardware support would have to be planned and implemented. Instead, our watchpoint tool operates in the spirit of ndb/NetSight [35], but it contains more features and does not duplicate and collect every data plane packet transmitted through OpenFlow-compliant switches. It has two operation modes: a proxy mode that monitors the control channel and a data plane mode that relies on switch capabilities to spot interesting packets. None of the operation modes modify the original OpenFlow entries injected by the controller and therefore avoid the overhead caused by traffic duplication. However, the data plane mode might result in large resource consumption if badly chosen watchpoints are triggered frequently.

As shown in *Figure 5.1*, the watchpoint tool transparently interposes on the control channel between switches and an SDN controller (or necessarily multiple controllers) to monitor and in special cases to modify the control traffic. This inter-layer, proxy role allows inspecting the behaviour of both control and datapath devices and carrying out the troubleshooting tasks without the need to modify these components. This kind of troubleshooting process requires reactive controller behaviour. In a SDN network with proactive controllers, this proxy operational mode can only observe a small part of the controller actions before incurring significant issues related to scalability. In this case, the data plane mode can be used to detour packets in “packet_in” OpenFlow messages to the watchpoint tool.

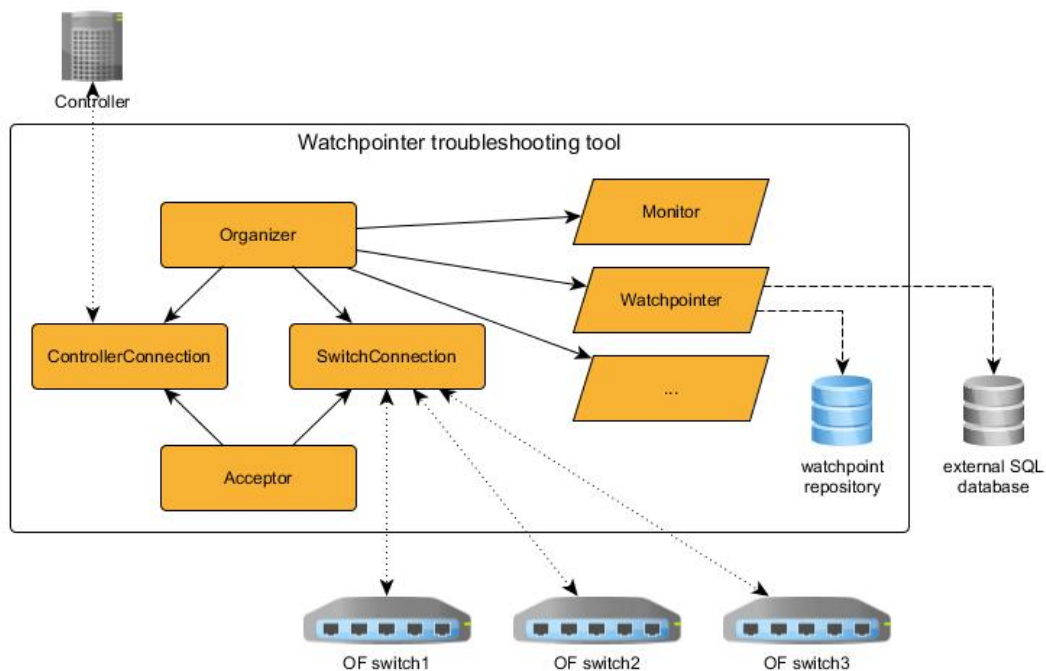


Figure 5.1: Schematic architecture of the watchpoint tool

The network watchpoints provide an opportunity for Service Developers or Operators to define OpenFlow-like filters to select relevant packets and automatically perform pre-defined actions collectively or on a per packet basis. Due to its monitoring functionality, matching packets could be filtered out and suspicious traffic could be detected in an operational network. Currently supported actions include *drop* (which enables implementing firewalls, intrusion detection systems, or intrusion prevention systems), *log*, *take snapshot* of flow tables and saving them into an SQL database, *query port* or other statistics and saving them, as well as *notify remote servers* through JSON/RPC calls.

This watchpoint-based functionality helps in the process of debugging an SDN network. Due to its design, the tool also allows extending its capabilities with additional functions. As an example, it can be integrated with network management frameworks or integrated development environments. In addition, this flexible approach also allows this tool to be used as a supplement

network element for extended security functions, i.e., one can implement a firewall-like filtering control subsystem or use it as a plug-in for intrusion detection and prevention systems.

port	Metadata	MACscr	MAC dst	EthType	VLANID	VLANprio	MPLSlabel	MPLSclass	IPsrc	IPdst	IPproto	IPToS	TCPsrc	TCPdst	DPID	Actions
(standard OpenFlow match fields)																

Figure 5.2: Structure of network watchpoint

Network watchpoints consist of three different parts, as shown in Figure 5.2. The first section contains the standard OpenFlow match fields. Since the match fields are standardized, the data plane operation mode can delegate the execution of the matching algorithm to switching hardware supporting the OpenFlow protocol. The second section of a watchpoint represents the relevant switch defined by a unique datapath ID (DPID). The last section represents the set of newly-defined troubleshooting actions. According to the OpenFlow matching fields, the DPID may contain a wildcarded value allowing a single entry to match multiple DPIDs. The programmatically defined actions allow integrating the watchpoint tool in the UNIFY Architecture and follow it with flexibility on its evolutionary path. The following basic troubleshooting actions are executed in the order of their definition:

- *No action* - Skip this execution step and process the next one. If there are no more actions defined, then it will send a message to the controller.
- *Drop* - Abort the execution and drop the packet immediately.
- *Log* - Write the triggered watchpoint along with other auxiliary information to the standard output (which could be a console or a graphical frontend).
- *Notify* - Notify the registered components provided by {URL | method | cookie} triplets via RPC messages. The prototype supports the JSON-RPC and XML-RPC standards.

In addition to these actions, more sophisticated actions could be developed as well. At this point in time, we implemented three more complex actions:

- *Snapshot* - Query the flow table entries from the connected OpenFlow switches and save the content into a previously defined SQL-based database. This function provides the opportunity to examine a snapshot of the network state offline with a third-party tool.
- *Port Stat* - Similar to *Snapshot*, but it collects port statistics and writes them to standard output instead of persistent storage.

- *Queue* - Enhanced version of *Log*. In addition of just writing the triggered action to standard output, the result is also stored in a queue. The queue can then be accessed by external tools through a JSON-RPC API call.

In line with the goals of the UNIFY Architecture, the watchpoint tool is based on a modular design ensuring flexibility, extensibility and reusability (*Figure 5.1*). The Acceptor sub-module is responsible for the basic management of the control channel and handles the join intentions from an SDN element. The SwitchConnection and ControllerConnection sub-modules handle the corresponding OpenFlow connection, which is organized and controlled by the Organizer sub-module. This Organizer module is also responsible for managing application-specific troubleshooting modules.

In the default, proxy operating mode, the watchpoint tool observes the control traffic passively and performs the matching process on every relevant OpenFlow message in order to find the longest matching watchpoint entry in the watchpoint repository (similar to a firewall). If there is no matching entry, then the tool forwards the captured OpenFlow message automatically to the controller. Because the tool checks only those messages that are relevant from a troubleshooting point of view, the overhead caused by the match operation could be kept low. However, this limits the amount of detectable incorrect actions. For example, an incorrectly defined flow table entry may not generate unexpected OpenFlow messages (Packet-In). Therefore such a problem does not occur in the control channel directly or at all, which makes finding the exact location of the problem difficult.

To overcome this shortcoming, we designed the data plane operation mode to take advantage of the internal matching functionality of OpenFlow switches. In this mode, the relevant messages can be checked against the watchpoint in the data plane. The watchpoint tool places the watchpoints as extra flow entries in the switch tables by translating them to Flow-Modification messages with highest priority, specific cookie value, and the OpenFlow action “send to controller”. In this case, if a packet matches the watchpoint, the generated OpenFlow message will be captured and identified by our tool and the watchpoint actions will be executed accordingly.

Although this approach achieves the data plane filtering without modification of the components, it means increased overhead for the network and interferes with the original communication. To meet the requirement of transparency, the tool performs the original OpenFlow actions with the corresponding OpenFlow messages. For this purpose, our tool follows the flow table changes by examining the control messages from both the controller and data plane component and recording the necessary changes. When a watchpoint placed in the data plane is triggered, the watchpoint tool searches for the appropriate actions and executes them. In summary, monitoring at data plane rates can be achieved with this additional operating mode in exchange for increased overhead, memory and CPU usage. The viability of this approach

depends on the number of data plane packets need to be transferred on the control channel, which in turn depends on the matching rules of the watchpoints.

We have developed our prototype atop the open source FlowVisor [36], using the OpenFlow protocol support (currently 1.0.0-rev1) with the message handling and parsing technique and the flow table tracking implementation. Moreover, we have reused the implementation of its advanced flow entry searching algorithm called “federated flowmap” with necessary modifications to fit to the watchpoint concept. The algorithm relies on a hash/trie-based indexing approach.

Currently, we see two ways for a VNF Developer to make use of network watchpoints. First, if a network function forwarding graph includes a control application, then the developer could monitor and debug the behaviour of that control application.

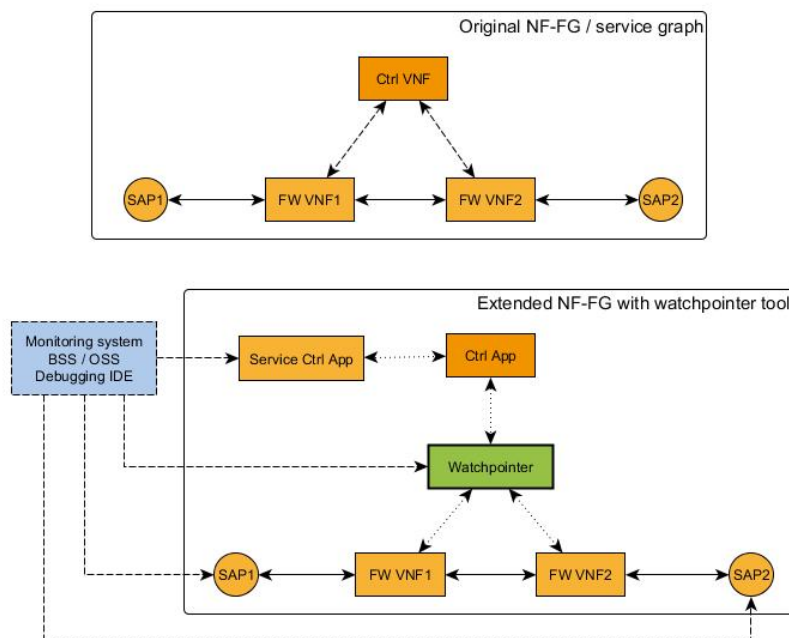


Figure 5.3: Use-case 1 (explicitly defined control app)

In Figure 5.3, we show a use case in which a developer has defined a simple forwarding service. In this case, the watchpoint tool inserts itself between the forwarding components and the controller application under test.

Second, even if the Forwarding Graph does not include a control application, the VNF Developer could set up watchpoints to verify the correctness of the traffic steering module of the UNIFY Architecture itself, or just to obtain usage statistics conveniently.

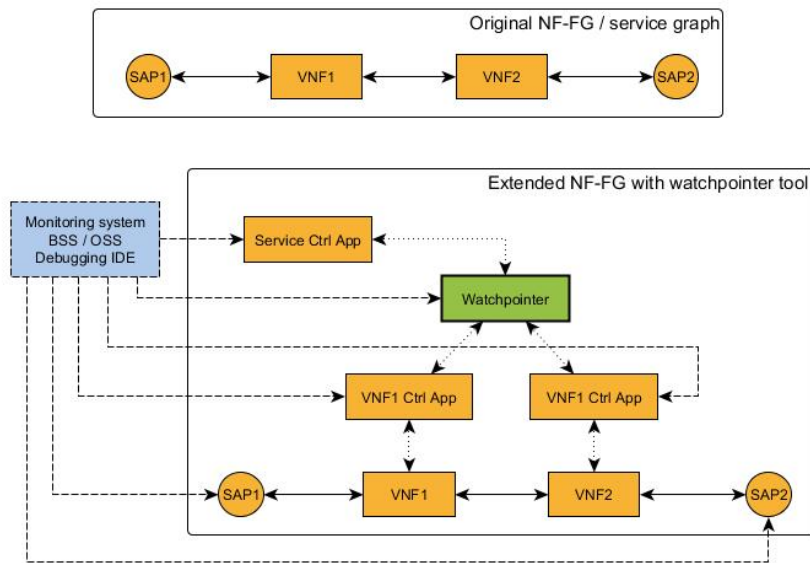


Figure 5.4: Use-case 2 (without explicitly defined control app)

In the second use case, the tool is put between the service control application and the VNF component, as shown in *Figure 5.4*, in order to monitor the traffic steering.

6 Troubleshooting

The SP-DevOps Troubleshooting process defined in D4.1 is based on workflows that orchestrate and automate capabilities developed by the Observability, Verification and VNF Development Support processes. As all the base capabilities for these processes are under development, it is too early to describe particular examples on how they could be assembled together for troubleshooting software-defined infrastructure.

Nevertheless, we note that the AutoTPG tool detailed in Section 4.2 could be employed in a reactive operation mode, meaning its capabilities could be triggered as part of an incident command check sheet attempting to rule out the existence of link failures, matching header failures, and the existence of forwarding loops. This is in contrast to the pro-active operation of the tool triggered by the SP-DevOps Verification process. The watchpoint tool described in Section 5.2 primarily as a reactive support for VNF Developers could be integrated into a workflow to provide observability capabilities beyond those available through a standard SP-DevOps Observability process, in particular through the *Snapshot* and *Queue* actions. In addition, the watchpoint tool allows defining specific filters for the messages to be investigated, thus offering flexibility to define per-workflow messages that are specific to the VNFs being troubleshooted and conserve compute resources by reducing the amount of matching that needs to be performed in the filters.

7 Integrated prototyping – DevOpsPro initial plans

Concerning the DevOps prototyping, the target of WP4 is to create one integrated prototype that demonstrates selected functionalities, tools and results developed within the work package. Moreover, we plan to work on several smaller scale prototypes combining two or three independently developed tools, and iterate further by taking into account experiences gained during the first phase of integration. Previous sections summarize the individual tools that partners are working on and some integration possibilities within each WP4 process. However, integration possibilities between tools focusing on different processes are yet to be considered.

Although integration approaches may differ in the first phase, T4.4 will coordinate the selection of common tools and help defining interfaces between modules. Therefore, major issues which cannot be treated with reasonable effort are not expected in this regard, as the integration procedure progressively continues.

Furthermore, prototypes of individual tools are planned to be evaluated on the same kind of software switch (OVS), controller framework (OpenDaylight), and cloud computing platform (OpenStack); the only exception of BME, who had already developed its tool of the Extensible Service ChAin Prototyping Environment (ESCAPE, [37]) using the POX OpenFlow controller framework. As the WP5 UN prototypes mature, we will consider the possibility to integrate the data plane functionality with xDPd [5] instead of OVS.

Finally, new separate communication interfaces, protocols, and/or channels need to be agreed on by the contributing partners so as to perform the integration of the various developed prototypes in an efficient manner. The detailed midterm plans of individual project partners are summarized below.

ACREO is currently focusing on the development of the monitoring messaging bus and the MEASURE language. We have a functional message bus at the moment that could already be used to assist the communication between components of an integrated prototype. However, the current implementation is quite basic, we plan to extend the current version with improvements on the internal protocol (between clients and brokers, and brokers to brokers) as well as to define a useful API for clients to connect to the bus, and support different messaging patterns such as reliable- and best-effort delivery. Additionally, partly as the development of the monitoring annotation language, we would like to define some standard message “envelopes” for sending measurement data, used to tag a measurement with metadata such as what type of measurement it is, where it originated, if and where to store it for longer times, etc. This development we plan to finish before the end of the project month 14.

The next steps for the monitoring messaging bus, in priority order, are as follows:

1. Define a simple client API for providing messaging services to various components.

2. Improve internal protocol for name distribution and support for reliable and unreliable messaging.
3. Define metadata format for measurement data.

For the monitoring language the current progress is a definition of the language itself and a formal description in terms of syntax and semantics, in the form of a grammar, with an accompanying parser. The implementation of the MEASURE consist of three main parts, the language itself, usage of the language within the orchestration and control environment, and aggregation points that can be placed in the system. The aggregation point is initially configured during the orchestration phase depending on the MEASURE. Once configured it receives measurement data and applies the defined aggregation zone arithmetic, and then performs the Action(s) for the current zone(s).

Planned next steps for the MEASURE language, in priority order, are:

1. Implement a basic aggregation point that can be configured, receive measurements, and perform actions.
2. Start investigating how the interaction with the orchestration layer may look like, this includes, e.g.:
 - a. How to determine where to place an aggregation point,
 - b. How to transform the MEASURE during function decomposition in the orchestration.

ACREO expect to be able to perform step 1 before the end of project month 14, and continue with step 2 later on.

BME's midterm prototyping actives focus on its multi-layer, multi-component debugging tool. At the end of month 15, BME plans to finish implementing a first version of the debugging tool. The tool will be able to help a developer to verify the correctness of a simple Service Graph containing a user-defined Network Function. Different small tools will be integrated into a common, uniform user interface of the *Emacs* editor. For example, the user will be able to (1) log into a remote host attached to the Service Access Point by the means of a couple keystrokes; (2) start a traffic source (or a simple ping) and the command prompt displayed within the editor; (3) look at the flow tables or port statistics of OpenFlow switches to verify traffic steering, and (4) attach Emacs's *gdb* debugger to the VNF running remotely in an execution environment. In the meantime, BME works on an interface that will simplify the integration of other tools with this debugging tool.

EAB will follow a Scrum-like development methodology and partition its WP4 prototyping activity into four *sprints*. The planned details of these sprints are as follows.

Sprint 1 (Start: project month 12, duration three weeks)

1. Adapt the loss measurement function to OpenDaylight and mininet.
2. Create backlog of items to be developed.
3. Validate the development environment by compiling and executing an example program using the monitoring interfaces in OpenDaylight.
4. Identify parts of the OpenFlow plugin implementation that need to be changed.
5. Document findings for and present for internal EAB review.

Sprint 2 (duration three weeks)

1. Select first items from backlog (examples: simple OpenFlow protocol extension for devolve action, policy-related configuration items for the OpenFlow agent on Open vSwitch, devolve action in Open vSwitch)
2. Develop and test items
3. Document findings for and present for WP4 review

Sprint 3 (duration three weeks)

1. Select next items from backlog (examples: add policies for devolve to Openflow extension, implement the loss measurement)
2. Develop and test items
3. Together with WP4 partners, define integration-related items for backlog (example: counters together with SICS, configuration together with TUB, etc)
4. Document findings for and present for WP4 review

Sprint 4 (duration three weeks)

1. Select next items from backlog (examples: add policies for devolve to OpenFlow extension, implement the loss measurement) including one integration item determined in Sprint 3
2. Develop and test items
3. Document findings for and present for WP4 review. First results from integration made available.

iMinds is focusing on developing AutoTPG for verification, troubleshooting, and observability techniques for link failures, matching header failures, packet loss, latency, forwarding loops errors, and bandwidth usage issues. iMinds plans to debug the network by: (1) transmitting test packets from the virtual machines (configured by OpenStack), (2) changing the headers according to the debugging conditions (using Click), (3) reading counters at the switches in the

data path (through OpenDaylight), and (4) receiving test packets in the virtual machines (configured by OpenStack) located in the network.

iMinds plans to follow following timelines for the prototyping:

Sprint 1 (Start: project month 12, duration four weeks)

1. Adapt the AutoTPG functional description to OpenDaylight, OpenStack, Click, and Virtual wall testbed at iMinds.
2. Identify parts of the Openflow plugin implementation that need to be changed.
3. Validate OpenStack by configuring virtual machines, validate OpenDaylight by reading counters and establishing flows in the switches, validate Click by changing headers.

Sprint 2 (duration four weeks)

1. Implementation of debugging techniques for link failures and matching header failures.
2. Creating a test environment on the testbed.
3. Calculation of Results.

Sprint 3 (duration four weeks)

1. Implementation of debugging techniques for packet loss, latency, bandwidth usage and protection path.
2. Creating a test environment on the testbed.
3. Calculation of Results.

Sprint 4 (four weeks)

1. Planning for Integration activity with other partners such as EAB (e.g., related to packet loss).
2. Develop and test items.
3. Document findings for and present for WP4 review. First results from integration made available.

POLITO plans to develop tools for deploy-time functional verification. The current activity is mainly focused on investigating the internals of VeriFlow and NetPlumber, the two state-of-the-art tools for SDN verification, specifically addressing OpenFlow-based systems. Depending on the results of this first activity, prototyping plans will be oriented to the definition of possible ways for extending the approaches of VeriFlow and/or NetPlumber to active network functions and non-pure OF environments. A central point of this activity will be the definition of possible approaches to model stateful VNFs. Hence, a key point will be the development of a proper

formal model for VNFs that takes VNF status into account. This model should then be integrated in one (or both) of the above tools. In case the results of the initial analysis will lead to the conclusion that the existing tools cannot be efficiently and/or effectively extended to support stateful VNFs, POLITO activities will mainly focus on the development of a brand-new verification tool that is able to satisfy our requirements.

Expected time to conclude the preliminary analysis is three months, while a first version of the extended tool is expected in six months. Subsequent months will be used for experimenting with the tool with the aim of improving its features and capabilities. In essence, it will be necessary to properly tune the VNF status formal model in order to ensure it can comprehensively handle active VNFs.

A further activity that will take place at the end of this first deployment phase will be the integration in the tool of proper graph-theoretic algorithms for the verification of graph topological properties, which can be directly applied to both the Service Graphs (SG) and to the Network Function Forwarding Graphs (NF-FG). This feature, currently not available in existing SDN verification tools, will provide a fast preliminary verification of the correctness of the deployed Service Chains, at least from a topological perspective.

SICS is currently developing methods for monitoring of link delay, loss, and utilization. Relevant parts of the code developed for evaluation of the link utilization monitoring method can be made available for integrated prototyping given suitable interfaces. Currently the link utilization monitoring method has been initially evaluated in a simulator framework written in Scala (based on the AKKA-framework). For the purpose of integration parts of the code would require modifications (in terms of porting or design of a wrapper interface), which at the moment need to be detailed in collaboration with relevant UNIFY partners. Specifically, SICS and EAB have discussed to investigate the possibilities of developing a joint first prototype within T4.4 based on partner individual prototypes for loss measurements (EAB) and link utilization monitoring (SICS). We see that the link utilization monitoring function will be mainly in focus for further integrated prototyping given a suitable use case, specifically if the plans on joint prototyping between SICS and EAB can be realized. Development of a first joint prototype is expected to start as soon as both the individual prototypes have matured during 2015.

TUB has the following two development plans. First, in the area of Troubleshooting / Network Tomography / Failover, TUB investigates on how to troubleshoot networks using a partial observability point deployment. Concretely, given a subset of SDN observability points, how would it be possible to 1. Quickly detect, and 2. Localize a link failure, and 3. Failover fast to an additional path. Second, TUB builds upon the Panopticon prototype (as mentioned in D4.1) and will leverage Spanning Tree Protocol (STP) event notifications to quickly failover, faster than what could be done with STP reconvergence. The work builds upon the theory of network tomography. A first prototype code is expected in 6 months. Second, concerning consistent network updates and inband mechanisms, we are investigating on how to build a proof-of-

concept in Mininet that shows how to consistently update network policies respecting waypoints and entire service chains. This complements our theoretical research disseminated by two papers at HotNets 2014.

OTE will participate in the prototyping by reviewing and providing feedback on tools and components developed to support SP-DevOps processes, periodically, focusing especially on network aspects and requirements. Moreover, OTE will participate in the discussion on the prioritization of tools and components for integration in the prototype, and in the evaluation of the integrated prototype by offering network resources and infrastructure such as a complete testbed comprising core, access, WLAN and mobile parts.

Deutsche Telekom (DT) and other partners within WP2 are currently discussing the methodology of the prototype integration. The aim of the discussion is to help partners identifying necessary parameters that are required for developing APIs between elements of individual prototypes. Furthermore, DT is working towards defining the detailed system architecture that is crucial to align prototyping work of different WPs. In general, DT's contributions to the prototype integration are three-fold. First, DT will incorporate individual workpackage prototypes (i.e., SPOOPro, DevOpsPro, and UNPro) into a solution that verifies and validates capabilities of prototypes. This will allow us to test existing prototypes before the actual integration happens. Second, DT will report on the achievements of UNIFY regarding IntPro which includes the integration effort, an overview of the components interworking, and evaluation of the performance. Finally, DT will debate on the way of the IntPro demonstration and document it.

Telecom Italia (TI) will participate in theoretical analysis and evaluations, providing the TI-Innovation perspective on the following aspects regarding SP-DevOps processes and tools developed by other WP4 partners:

- Early detection and control of “instabilities” in SDN-NFV-like Carriers Networks (with automated configurations and where functions are like transactions allocated and move dynamically);
- Implications of Consistency, Availability, Partitioning (CAP) Theorem in managing the “states” of virtual and real resources;
- Issues about the overall latency assessment in e2e service chains or VNF transaction;
- Evolution from Telecommunications Management Network (TMN) towards a OS-like approach

Conclusion

In this milestone document, we outlined the progress on integrating the SP-DevOps concept with the UNIFY Architecture, along with detailing steps forward taken towards the development of particular tools aimed at monitoring and verification in software-defined telecom infrastructure. We also present the initial plans for prototyping activities, aimed at first developing initial tools and then integrating selected functionality into the SP-DevOps Toolkit and the UNIFY integrated prototype.

The SP-DevOps concept was evolved by defining a set of APIs that complement all the reference points defined in the UNIFY Architecture. Each API defines a small set of function calls allowing for monitoring information to be requested and transferred between layers of the UNIFY Architecture in line with the SP-DevOps Observability and VNF Development Support processes. The APIs also support the SP-DevOps Verification process through a dedicated function call. The major components of the SP-DevOps Toolkit were outlined, namely annotations for monitoring and verification capabilities and data that could be embedded in the Network Function Forwarding Graph description in collaboration with WP3, as well as a set of tools addressing particular problems identified as research challenges in D4.1. The Observability Points were made more concrete with respect to possible ways of integrating them onto the Virtual Switch Environment of the WP5 Universal Node through the use of OpenFlow experimental actions. Steps forward were also presented towards defining a common, scalable communication framework for monitoring data based on ZeroMQ messaging, as well as aggregation capabilities given by extensions of the VirtuCast algorithm and consistent configuration through applying the WayUp algorithm.

A number of tools are developed by the partners and discussions are ongoing with respect to their availability in the SP-DevOps Toolkit. Some tools may be open sourced, while others are expected to remain proprietary. Mapping of all the tools visually as NF-FGs is presented as an intermediary step towards a representation in a combination of monitoring language and NF-FG formal monitoring descriptions that will be available later in the project. Representations of all the tools as Observability Points and Monitoring Functions are also included with the aim to show the coherence of overall approach and open the way for integration in the SP-DevOps Toolkit.

Statistics-based counters enhance the capabilities of Universal Nodes to report on the changing nature of network traffic. These counters could be extended to metrics related to compute resources, a step forward in achieving a unified resource view in the UNIFY production environment. Methods for enhancing the estimates for packet loss and latency in software-defined networks were presented, addressing known pain points such as the expected preponderance of aggregated flow descriptors in carrier networks and creating opportunities to reduce the measurement overloads on both the data and control planes. The multi-purpose AutoTPG and watchpoint tools provide additional support to both Operators and VNF Developers

by improving the observability of the control plane (the watchpoint tool) and by enabling a more comprehensive verification of the forwarding properties of the network in terms of flow table contents and actions associated to the installed flows (AutoTPG). An incipient network debugger tool allows the VNF Developers to automatically connect a well-known software debugger to VNF instances from Emacs, an environment which is familiar to people developing scripts in the Linux operating system. Results from a comprehensive investigation show the overhead of state-of-the-art formal verification tools for software-defined networks, and determined that their major limitation in a UNIFY production environment relates to the inability to verify virtual network functions. Initial ideas on directions to be taken for addressing this limitation, along with other disadvantages of existing tools, are also presented. Annex 2 presents a mapping of the progress reported in this milestone towards the objectives specified in the Description of Work document. Although a particular result may contribute towards achieving several objectives, the mapping focuses on the objective that in our opinion is addressed by the core of a particular result.

Finally, initial plans for prototyping of key functionality were outlined. Although the plans concentrate on activities from individual partners, two avenues for integration and strengthened collaboration were opened through step-wise integration between two partners initially and then deployment and evaluation on infrastructure from operator labs. Within six months from the submission of this milestone report, we expect significant progress on the integration of individual partner developed functionality. The participants from the three telecom operators will act as soundboards in the evaluation of the functionality developed within the Workpackage and potentially host parts of DevOpsPro in their own lab environments for experiments.

List of abbreviations and acronyms

Abbreviation	Meaning
CP	Connection Point
DP	Data Plane
LCP	Local Control Plane
MF	Monitoring Function
MF CP	Monitoring Function Control Plane
MF DP	Monitoring Function Data Plane
NF-FG	Network Function Forwarding Graph
OP	Observability Point
UN	Universal Node
URM	Universal Resource Manager
VNF EE	Virtual Network Function Execution Environment
VSE	Virtual Switching Environment
NFV	Network Function Virtualization
OAM	Operations, Administration and Maintenance
SAP	Service Access Point
SDN	Software Defined Network
VNF	Virtual Network Function

8 Bibliography

- [1] "D4.1 Initial requirements for the SP-DevOps concept, universal node capabilities and proposed tools," FP7 UNIFY project, 2014.
- [2] "D2.1 Initial Architecture Framework," FP7 UNIFY project, 2014.
- [3] "D2.2 Final Architecture," FP7 UNIFY, 2014.
- [4] "D3.1 Programmability framework," FP7 UNIFY Project, 2014.
- [5] "D5.2 Universal Node Interfaces and Software Architecture," FP7 UNIFY project, 2014.
- [6] ETSI, "Network Function Virtualization (NFV) Management and Orchestration V0.6.1 (draft)," July 2014. [Online]. Available: http://docbox.etsi.org/ISG/NFV/Open/Latest_Drafts/NFV-MAN001v061-20management and orchestration.pdf. [Accessed October 2014].
- [7] TMForum, "TR198, Multi-Cloud Service Management Pack - Simple Management API (SMI) Developer Primer and Code Pack, Release 1.0," TMForum, 2014.
- [8] "Transforming to DevOps with Junos OS," 2014. [Online]. Available: <http://www.juniper.net/assets/us/en/local/pdf/whitepapers/2000586-en.pdf>. [Accessed November 2014].
- [9] D. Roberts, "Why "Enterprise DevOps" Doesn't Make Sense," 10 November 2014. [Online]. Available: <http://devops.com/features/enterprise-devops-doesnt-make-sense/>. [Accessed 10 November 2014].
- [10] "Job queues, message queues and other queues. Almost all of them in one place.," [Online]. Available: <http://queues.io/>. [Accessed October 2014].
- [11] "RabbitMQ," [Online]. Available: <http://www.rabbitmq.com>. [Accessed October 2014].
- [12] "ZeroMQ," [Online]. Available: <http://www.zeromq.org>. [Accessed October 2014].
- [13] M. Rost, "Optimal Virtualized In-Network Processing with Applications to Aggregation and Multicast," TUB, Berlin, 2014.
- [14] M. Rost and S. Schmid, "Virtucast: Multicast and aggregation with in- network processing," in *Principles of Distributed Systems, volume 8304 of Lecture Notes in Computer Science*, Springer International Publishing, 2013, p. 221-235..
- [15] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger and D. Walker, "Abstractions for Network Update," in *Proceedings of the ACM SIGCOMM 2012*, New York, NY, USA, 2012.

- [16] R. Mahajan and R. Wattenhofer, "On Consistent Updates in Software Defined Networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks NotNets-XII*, New York, NY, USA, 2013.
- [17] A. Ludwig, M. Rost, D. Foucard and S. Schmid, "Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies," in *Proceedings of the Thirteenth ACM Workshop on Hot Topics in Networks HotNest-XIII*, New York, NY, USA, 2014.
- [18] R. Presuhn, "Management information base (MIB) for the simple network management protocol (SNMP)," RFC 3418, 2002.
- [19] J. Postel, "Internet control message protocol," RFC 792, 1981.
- [20] "Introduction to Cisco IOS NetFlow - A Technical Overview," May 2012. [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.pdf. [Accessed October 2014].
- [21] M. Wang, B. Li and Z. Li, "sFlow: Towards resource-efficient and agile service federation in service overlay networks," in *Proc. of the 24th IEEE International Conference on Distributed Computing Systems*, 2004.
- [22] P. Kreuger and R. Steinert, "Scalable in-network rate monitoring," in *submitted to IM 2015*.
- [23] R. Steinert, "Probabilistic Fault Management in Networked Systems," KTH, Stockholm , 2014.
- [24] "OpenFlow Switch Specification," October 2013. [Online]. Available: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf>. [Accessed October 2014].
- [25] A. Khurshid, W. Zhou, M. Caesar and P. B. Godfrey, "VeriFlow: verifying network-wide invariants in real time," in *HotSDN '12*, 2012.
- [26] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown and S. Whyte, "Real Time Network Policy Checking using Header Space Analysis," in *NSDI13*, 2013.
- [27] P. Kazemian, G. Varghese and N. McKeown, "Header Space Analysis: Static Checking For Networks," in *NSDI12*, 2012.
- [28] P. Kazemian, "Hassel-Public," 2014. [Online]. Available: <https://bitbucket.org/peymank/hassel-public/wiki/Home>.
- [29] A. Khurshid, W. Zhou, M. Caesar and P. B. Godfrey, "VeriFlow Source Code Release Form," [Online]. Available: <http://web.engr.illinois.edu/~khurshi1/projects/veriflow/veriflow-code-release.php>. [Accessed Settembre 2014].

-
- [30] "Mininet: Rapid prototyping for software defined networks.," [Online]. Available: <http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet..>
 - [31] A. Panda, O. Lahav, K. Argyraki, M. Sagiv and S. Shenker, "Verifying Isolation Properties in the Presence of Middleboxes," September 2014. [Online]. Available: <http://arxiv.org/abs/1409.7687>. [Accessed November 2014].
 - [32] H. Zeng, P. Kazemian, G. Varghese and N. McKeown, "Automatic Test Packet Generation," in *8th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Nice, 2012.
 - [33] S. Kolahi, S. Narayan, D. Nguyen and Y. Sunarto, "Performance Monitoring of Various Network Traffic Generators," in *Proceedings of 13th International Conference on Computer Modelling and Simulation (IEEE UKSim)*, 2011.
 - [34] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.
 - [35] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières and a. N. McKeown, "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.
 - [36] "FlowVisor," [Online]. Available: <http://onlab.us/flowvisor.html>.
 - [37] A. Csoma, B. Sonkoly, L. Csikor, F. Nemeth, A. Gulyas, W. Tavernier and S. Sahhaf, "ESCAPE: Extensible Service ChAin Prototyping Environment using Mininet, Click, NETCONF and POX," in *SIGCOMM'14*, Chicago, 2014.

Annex 1 SP-DevOps components for UNIFY Architecture Interfaces

8.1 Extensions to the SI-Or interface

The following functions are proposed to be implemented through the SI-Or interface in order to support the SP-DevOps Observability, Verification, Troubleshooting and VNF Development Support processes. The NF-FG was defined in [4].

GetPerformanceValues - collects the latest values measured for a list of performance metrics associated to the NF-FG

Input Parameters: NF-FG identifier (mandatory), list of metrics for which the values are to be returned (optional)

Output Parameters: Set of metric-value pairs measured during the last interval for all the metrics associated with the NF-FG (in case a list of metrics was not provided) or only for those metrics specified in the optional input parameter.

SetupNotification - configures automated notifications regarding performance metrics or state transitions associated with the NF-FG

Input Parameters: NF-FG identifier (mandatory), listener (mandatory), list of metrics or state transitions for which the values are to be returned (optional), notification criteria (optional). The listener is an identifier for an entity which would receive the notification. The listener must be registered with the RegisterListener call beforehand. The notification criteria allow specifying, within certain limits, filtering capabilities for the issuing of notifications. For example, a notification could be sent only for a pre-determined number of times within a pre-defined time window, or be generated only when the value of a particular metric rises above or descends below a certain threshold (hence no continuous updates are generated at pre-defined time intervals), etc.

Output Parameters: Boolean value reflecting whether the setup was successful (True) or not (False).

Notify - provides automated notifications regarding performance metrics or state transitions associated with the NF-FG as configured via the SetupNotification function

Input Parameters: None

Output Parameters: NF-FG identifier (mandatory), metric or state identifier (mandatory), Type-Length-Value (TLV) triplet representing the information to be notified to the listener(s).

RegisterListener / UnregisterListener - makes a change to the NF-FG to include a communication endpoint with an entity such that this entity is able to invoke the GetPerformanceValues or Verify function calls on the interface and receive the results, or

receive notifications through the Notify function. Such entity could be, for example, a ControlApp associated with functions such as scaling in the UNIFY Orchestration, or an OSS of the service provider that performs customer experience management functions.

Input Parameters: NF-FG identifier (mandatory), NF-FG components for adding or removing an additional communication point

Output Parameters: Boolean value that indicates whether the operation was successful (true) or not (false)

8.2 Extensions to the Co-Rm interface

The following functions are proposed to be implemented through the Co-Rm interface in order to support the SP-DevOps Observability, Verification, Troubleshooting and VNF Development Support processes:

GetPerformanceValues - collects the latest values measured for a list of performance metrics associated with the virtual resource allocated to an NF-FG component

Input Parameters: Virtual resource identifier (mandatory), list of metrics for which the values are to be returned (optional). The virtual resource identifier could for example be a Network Element Id (NE Id) as defined in [4].

Output Parameters: Set of metric-value pairs measured during the last interval for all the metrics associated with the virtual resource (in case a list of metrics was not provided) or only for those metrics specified in the optional input parameter

SetupNotification - configures automated notifications regarding performance metrics or state transitions associated with the virtual resource

Input Parameters: Virtual resource identifier (mandatory), listener (mandatory), list of metrics or state transitions for which the values are to be returned (optional), notification criteria (optional). The virtual resource identifier could for example be a Network Element Id (NE Id) as defined in [4]. The listener is an identifier for an entity which would receive the notification. The listener must be registered with the RegisterListener call beforehand. The notification criteria allow specifying, within certain limits, filtering capabilities for the issuing of notifications. For example, a notification could be sent only for a pre-determined number of times within a pre-defined time window, or be generated only when the value of a particular metric rises above or descends below a certain threshold (hence no continuous updates are generated at pre-defined time intervals), etc.

Output Parameters: Boolean value reflecting whether the setup was successful (true) or not (false).

Verify - performs one or more verification actions on some or all the verifiable properties associated with virtual resources

Input Parameters: virtual resource identifier (mandatory), list of properties that need to be verified (optional), verbosity level (optional). The virtual resource identifier could for example be a Network Element Id (NE Id) as defined in [4]. The list of properties that could be verified include consistency of rules in Openflow switch tables, the existence of forwarding black holes in OpenFlow switch tables, which ports are part of a particular VLAN, the architecture type (X86 or ARM, for example) of a processor where the virtual compute resource for a VNF container is executed, etc. A comprehensive list of properties that need to be verified is dependent on the capabilities of the virtual resources. The verbosity level is an optional parameter that controls the detail included in the results. For example, with a verbosity level set to "low", a verification of the absence of loops could return simply "pass" or "fail", while the same verification action could in addition return the identifiers of the forwarding resources that are part of the detected loop.

Output Parameters: list of results from the verification actions performed on the service.

Notify - provides automated notifications regarding performance metrics or state transitions associated with the virtual resource as configured via the SetupNotification function

Input Parameters: None

Output Parameters: Virtual resource identifier (mandatory), metric or state identifier (mandatory), Type-Length-Value (TLV) triplet representing the information to be notified to the listener(s). The virtual resource identifier could for example be a Network Element Id (NE Id) as defined in [4].

RegisterListener / UnregisterListener - registers an entity to receive notifications through the Notify function. It assumes that the communication path between the Resource and the Listener was pre-setup through the NF-FG. Such entity could be, for example, a ControlApp associated with functions such as scaling in the UNIFY Orchestration, or an OSS of the service provider that performs customer experience management functions.

Input Parameters: Virtual resource identifier (mandatory), listener identifier (mandatory). The virtual resource identifier could for example be a Network Element Id (NE Id) as defined in [4].

Output Parameters: Boolean value that indicates whether the operation was successful (true) or not (false)

8.3 Extensions to the Or-Ca and Ca-Co interfaces

The following functions are proposed to be implemented through the Or-Ca interface in order to support the SP-DevOps Observability, Verification, Troubleshooting and VNF Development Support processes. The NF-FG was defined in [4].

GetPerformanceValues - collects the latest values measured for a list of performance metrics associated with the NF-FG

Input Parameters: NF-FG identifier (mandatory), list of metrics for which the values are to be returned (optional)

Output Parameters: Set of values measured during the last interval for all the metrics associated with the NF-FG (in case a list of metrics was not provided) or only for those metrics specified in the optional input parameter

Verify - performs one or more verification actions on some or all the verifiable properties associated with a NF-FG

Input Parameters: NF-FG identifier (mandatory), list of properties that need to be verified (optional), verbosity level (Optional). Such list of properties that could be verified include packet transit through the NF-FG, reachability of various NF-FG components, the lack of loops or maximum number of times a particular component of a NF-FG is to be traversed, etc. A comprehensive list of properties that need to be verified is dependent on the NF-FG. Work on formal descriptions for such properties is at too much of an early stage to be reported in this document. The verbosity level is an optional parameter that controls the detail included in the results. For example, with a verbosity level set to "low", a verification of the absence of loops could return simply "pass" or "fail", while the same verification action could in addition return the identifiers of the NF-FG components that are part of the detected loop.

Output Parameters: list of results from the verification actions performed on the service.

SetupNotification - provides automated notifications regarding performance metrics associated with the NF-FG

Input Parameters: NF-FG identifier (mandatory), listener (mandatory), list of metrics or state transitions for which the values are to be returned (optional), notification criteria (optional). The listener is an identifier for an entity which would receive the notification. The listener must be registered with the RegisterListener call beforehand. The notification criteria allow specifying, within certain limits, filtering capabilities for the issuing of notifications. For example, a notification could be sent only for a pre-determined number of times within a pre-defined time window, or be generated only when the value of a particular metric rises above or descends below a certain threshold (hence no continuous updates are generated at pre-defined time intervals), etc.

Output Parameters: Boolean value reflecting whether the setup was successful (true) or not (false).

Notify - provides automated notifications regarding performance metrics or state transitions associated with the NF-FG as configured via the SetupNotification function

Input Parameters: None

Output Parameters: NF-FG identifier (mandatory), metric or state identifier (mandatory), Type-Length-Value (TLV) triplet representing the information to be notified to the listener(s).

RegisterListener / UnregisterListener - makes a change to the NF-FG to include a communication endpoint with an entity such that this entity is able to invoke the GetPerformanceValues or Verify function calls on the interface and receive the results, or receive notifications through the Notify function. Such entity could be, for example, a ControlApp associated with functions such as scaling or optimization in the UNIFY Orchestration.

Input Parameters: NF-FG identifier (mandatory), NF-FG components for adding or removing an additional communication point

Output Parameters: Boolean value that indicates whether the operation was successful (true) or not (false)

Annex 2 Mapping of progress reported towards WP4 objectives

Table 2: Short summary of progress reported in this milestone mapped towards DoW objectives

DoW WP4 Objective	Progress reported in this milestone
O4.1 Evaluate and demonstrate, in an agile manner, the SP-DevOps concept for selected scenarios, including the development of the Service Provider DevOps prototype (DevOpsPro)	Description of individual partner prototyping and evaluation activities, along initial plans for integration in section 7
O4.2 Define conditional observability points located on Universal Nodes and develop an automated approach for deploying them consistently	Evolution of Observability Points regarding the integration in the UNIFY architecture in section 2.2 Configuration update consistency considerations in section 3.4
O4.3 Develop scalable service monitoring approaches, adapted to software-defined networks, that are efficient in reducing the number of manual diagnosing steps and amount of observation data transiting on the network	Messaging for Observability Points based on the ZeroMQ framework in section 3.1 In-network data aggregation by enhancing the VirtuCast algorithm in section 3.3 Two specific capabilities for scalable monitoring of link utilization, delay and packet loss in section 3.5
O4.4 Design methods for verifying service chain functionality at runtime and locating service chain faults	Investigation on the scalability and features offered by the NetPlumber and VeriFlow tools, as well as considerations for upcoming work in section 4.1 AutoTPG tool for verifying SDN forwarding plane in section 4.2 Watchpoint tool for visibility onto the SDN control plane in section 5.2
O4.5 Enable automatic definition of workflows for verification and activation tests for dynamic service chains	High-level specification of interfaces for integrating observability and verification within the UNIFY Architecture in section 2.1 and Annex 1 Considerations on describing monitoring functions through a formal language in section 3.2
O4.6 Enable the possibility to verify service chains within the limit of one development cycle	Planned future steps of the investigation on NetPlumber and VeriFlow in section 4.1

	Multi-layer debugger tool development in section 5.1
--	--