ScuDo

Scuola di Dottorato ∽ Doctoral School

WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation

Doctoral Program in Computer and Control Engineering (29thcycle)

Verification and Configuration of Software-based Networks

By

Serena Spinoso

******

Supervisor(s):

Prof. Riccardo Sisto

Doctoral Examination Committee:

Dr. Luca Durante, Referee, IEIIT-CNR
Prof. Adlen Ksentini, Referee, Eurecom
Prof. Eduardo Jacob, University of the Basque Country
Prof. Antonio Lioy, Politecnico di Torino
Prof. Bartolomeo Montrucchio, Politecnico di Torino

Politecnico di Torino
2017

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Serena Spinoso
2017

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

A te, papà.
Grazie di tutto.

Acknowledgements

I was never good at expressing my feelings and gratitude neither verbally nor in writing, but I'm trying to do it here because I would like to sincerely thank those people who have accompanied me in this trip that has been my PhD.

First of all, I would like to show my gratitude to my supervisor, Prof. Riccardo Sisto, who has driven and helped me in the last three years with kindness and availability, giving me so many opportunities.

I would like to thank also all the people that I met at the Networking Group, with a special mention for Ivano, Matteo, Marco, Roberto, Amedeo and Francesco that have made this experience much more enjoyable and gratifying.

Last but not least, I want to show my appreciation to my family, which has always supported and endured me (and unfortunately for them, they will have to continue to do it), especially Fulvio, without whom I would not be here, writing the acknowledgements for this thesis.

Thanks,
Serena

Abstract

The innovative trends of Network Function Virtualization (NFV) and Software Defined Networking (SDN) have posed never experienced opportunities in productive environments, like data centers. While NFV decouples software implementation of the network functions (e.g., DPI and NAT) from their physical counterparts, SDN is in charge of dynamically changing those functions to create network paths. One new opportunity of such *Software-based networks* is to make the network service-provisioning models more flexible, by enabling users to build their own *service graphs*: users can select the Virtual Network Functions (VNFs) to use and can specify how packets have to be processed and forwarded in their networks. In particular, this PhD thesis spans mostly topics related to the verification and configuration of service graphs.

For what concerns the challenges of network verification, our aim is to explore strategies that overcome the limitations of traditional techniques, which generally exploit complex modelling approaches and takes considerable verification times.

Thus we envision for verification techniques that are based on non-complex modelling approaches in order to be much more efficient than existing proposals. Under these conditions, such novel approaches may work at run-time and, in particular, may be performed before deploying the service graphs, in order to avoid unexpected network behaviours and detect errors as early as possible.

Another requirement is that verification should take a reasonable amount of time from a VNF Orchestrator point of view, with fair processing resources (e.g. CPU, memory and so on). This is because we are in the context of flexible services, where the reconfiguration of network functions can be frequently triggered, both in case of user request and in case of management events.

The first contribution of this thesis lays on the service graphs specification by means of *forwarding policies* (i.e, a high-level specification of how packet flows are

forwarded). While the majority of the SDN verification tools operate on OpenFlow configurations, we have defined a formal model to detect a set of *anomalies* in forwarding policies (i.e., erroneous specifications that may cause misleading network conditions and states). The key factors that distinguish our work from existing approaches are both an early detection of policies anomalies (i.e., before translating such policies into OpenFlow entries), in order to speed up the fixing phase, without even starting service deployment, and a scalable approach that achieves verification times in the order of milliseconds for medium- large- sized networks.

Another advancement in network verification has been the possibility to verify networks including *stateful VNFs*, which are functions that may dynamically change the forwarding path of a traffic flow according to their local algorithms and states (e.g., IDSs). Our second contribution is thus a verification approach that models the network and the involved (possibly stateful) VNFs as a set of FOL formulas. Those formulas are passed to the off-the-shelf SMT (Satisfiability Modulo Theory) solver Z3 in order to verify some reachability-based properties. In particular, the proposed solution has been implemented in a tool released under the AGPLv3 license, named VeriGraph, which takes the functional configurations of all deployed VNFs (e.g., filtering rules on firewalls) into account to check the network. The adopted approach achieves verification times in the order of milliseconds, which is compliant with the timing limitations needed by a VNF Orchestrator.

Finally, for what concerns the configuration of VNFs, service graph deployment should include a strategy to deploy VNF configurations in order to fix bugs in case of verification failures. Here, we have to face several challenges like the different ways a network function may require for being configured (REST API, CLI, etc...) and the configuration semantic that depends on the function itself (e.g., router parameters are clearly different from firewall ones).

We conclude this thesis by proposing a *model-based configuration* approach, which means defining a representation of the main configuration parameters of a VNF. This VNF model is then automatically processed by further software modules in the VNF architecture to translate the configuration parameters into a particular format required by a VNF and to deliver the produced configuration into the VNF following one of the configuration strategies (e.g., REST, configuration file, etc.) already supported by the function. The achieved results of this last work, *w.r.t.* the current state of the art, are the exploitation of a model-driven approach that achieves a higher flexibility and the insertion of non-VNF-specific software modules to avoid changes in the VNF implementation.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The innovative trends of Network Function Virtualization (NFV) and Software Defined Networking (SDN) have posed never experienced opportunities in productive environments, like data centers.

NFV[1] aims at decoupling software implementation of the network functions, like routers, firewalls, Deep Packet Inspections and others, from their physical counterparts. This means that network functions are transferred from dedicated hardware appliances to software-based applications, named Virtual Network Functions (VNFs), running on commercial off-the-shelf (COTS) equipment. Thanks to this paradigm, VNFs can be instantiated in various locations such as data centers, network nodes, and end-user premises as the network requires.

SDN[2] [1], instead, separates the control plane from the data plane, which means that network devices have to simply forward traffic while the control logic is centralized in a single software program, named Controller. This separation has made the network more programmable and easy to manage: providers have to instruct a single software model to manage the whole network and to apply the desired network behaviour. In particular, the SDN Controller can be programmed to instantiate network paths where packets are forwarded through an ordered set of network functions before being delivered to the final destination. Such network paths are known in the literature as *Service Function Chains* (SFCs) [2].

[1] http://www.etsi.org/technologies-clusters/technologies/nfv

[2] https://www.opennetworking.org/sdn-resources/sdn-definition

Such paradigms are part of the recent trend named *Softwarization*, which is having a significant impact on the way network service are deployed and managed by providers. The Softwarization of the network has forsaken the usual service provisioning model that was strictly intertwined with the physical network topology and based on the typical switching and bridging solution, in favour of new chain-based models.

Recent research activities, in fact, have enabled tenants to define multiple SFCs that could be also overlapped, creating a sort of forwarding graph where traffic can follow different paths according to network functions configuration and packets content. Thus, this novel chain-based network service enables users to build their own networks (named *service graphs*). During the service graph specification phase, the final users can select the VNFs to use and can specify how packets have to be processed and forwarded in the graph.

From the provider perspective, the deployment of a service graph needs a NFV architecture and many components to manage the VNF instance, the resource assignment and the whole service lifecycle. One of the main components of this architecture is the VNF Orchestrator, which is responsible for the management of the network service lifecycle and of the global resources.

In the context of the Software-based networks (i.e., NFV/SDN-based), several research problems can be addressed like the optimization of VNF placement, the enforcement of security-related constraints in the service graph, or the enforcement of bandwidth constraints. However, this PhD thesis spans mostly topics related to the verification of Software-based networks and their configuration.

In particular, we will consider the context of flexible network services, where the reconfiguration of the involved VNFs can be frequently triggered, both in case of user request and in case of management events. Under these conditions, on one side, mechanisms for verification and properties checking are needed because providers need a high level of assurance of the network service correctness, before deploying it; on the other side, providers should also take care of implementing a seamless strategy to deploy VNF configurations in order to complete the service request and/or eventually to fix bugs in case of verification fails.

In order to improve the state of the art in these directions, we present in this dissertation our contributions to the network verification and configuration. In particular, the remainder of this thesis is organized as follows:

- Chapter 2 presents a solution of *early-verification*. We aim to check the presence of anomalies and faults in the user's request expressed in form of *forwarding policies*, which are an high-level specification of traffic forwarding translated into low-level configurations to install in the SDN network. This work exploits a formal model to represent the forwarding policy and to check a set of anomalies before configuring the network;

- Chapter 3, instead, proposes a way to model both stateless and stateful VNFs (i.e. network functions that forward packets based on both packet content and traffic history). This approach has been implemented in an open source tool that allows to check the satisfaction of reachability-based invariants in service graph requests and to stop the graph deployment in case of verification failure;

- The last part of this dissertation (Chapter 4) focuses on the configuration of Software-based networks by proposing an effective solution for installing configuration parameters into VNFs (e.g., blacklists into DNS filters) with a model-based approach;

- Chapter 5 finally concludes this thesis by presenting possible improvements of our contributions to the state of the art.

Chapter 2

Detecting Anomalies in Service Function Chains

In this thesis, we mean verification as the process used to check the correctness of computer systems before putting them into use. Besides, the verification process can have a formal approach, i.e. it may rely on formal methods and mathematical reasoning to perform its goal. In particular, formal verification has been applied firstly to check the correctness of hardware and it is now increasingly used in the software development process. In this dissertation, we are interested in the application of formal verification in the networking field and how the literature has addressed the problem in order to improve the state of the art with novel proposals.

There has been much work on checking network protocols and their implementations, but until recently almost none on verifying a given network configuration. Recently, indeed, many approaches and tools have been applied to verify if the current or proposed configuration obeys various important invariants defined by the provider in order to guarantee that the network will operate correctly. Examples of such invariants are absence of routing loops or black holes, network node reachability, satisfaction of security-related policies (e.g., node isolation and traversal) and others.

In this thesis, we mostly focus on those mechanisms for verification and property checking applied in the context of SDN/NFV-based networks. With respect to this topic, the majority of the proposed verification tools and approaches operate on network configurations represented by the forwarding tables installed into the routers and switches. Moreover, the forwarding decisions of a SDN-controlled router/switch are commonly dictated by OpenFlow, the standard implementation of the SDN paradigm [3].

Current SDN-oriented tools verify if some invariants hold by checking only the low-level configurations (e.g. OpenFlow) of the SDN switches, causing a late detection of errors and faults in the service graph requests. Due to the high agility, flexibility and programmability of Telecommunications infrastructures, providers need novel verification approaches that, on one side, overcome the limitations of traditional model checking techniques, which may fall in out of memory and time in case of complex scenarios, and, on the other side, speed up the fixing phase in case of verification failures. In order to improve the state of the art in these directions, we present in this chapter a scalable solution to verify Software-based networks before they are configured.

2.1 Problem statement and contributions

We recall that the Service Function Chaining (SFC) concept [4] consists in instantiating an ordered sequence of network functions, and consequently steering a particular portion of packets (e.g., the ones of a particular user) through the deployed chain. This new paradigm has improved the network services offered to end-users, who can express how their own traffic should be processed inside the network. However SFC services have also introduced additional complexity and many challenges in flow management.

SDN has provided the means to address such challenges and to reduce the complexity of managing networks. In this context, OpenFlow is the first SDN standard, which expresses routing state as a set of `<match,action>` entries in FlowTables, named *flow entries*. When a network switch receives a packet with headers matching the match entry, that packet is subject to the specified action (e.g., "`forward to a specific port`", "`drop packet`", etc.), otherwise it is forwarded towards the SDN Controller. The Controller, then, will install the appropriate flow entry in the network to manage that packet.

A proposed strategy to simplify SFC instantiation is to enable SDN/OpenFlow Controllers to configure the network directly by means of forwarding policies. By *forwarding policy* we refer to a network policy [1] that specifies how traffic should traverse the instantiated chains in a high-level and abstract way.

In literature, many policy-oriented languages have been proposed to program SDN networks. Generally such languages provide high-level constructs for defining how packets must be classified and forwarded. Examples are Merlin [5] and FatTire [6]. After the definition phase, a forwarding policy is translated into the most suitable set of low-level rules, like flow entries in OpenFlow FlowTables [3]. The SDN Controller will install the translated flow entries into the network switches in order to instantiate the desired service chains.

Of course, errors and ambiguity in policy specifications can lead to faulty network configurations. For example, a forwarding policy could generate conflicting flow entries in OpenFlow switches. Let us consider the case of a switch FlowTable containing two conflicting entries that manage the same traffic flow, but that assign this flow different actions like "`drop`" and "`flood`". Even tough OpenFlow adopts a resolution strategy based on priorities among flow entries, the higher-priority entry may not be the desired one for managing that traffic flow. At run-time, faults in network configuration may be present in managing traffic flows with opposite actions to the entries which should take precedence in the FlowTable.

Our contribution mainly aims at preventing these problems by defining a formal model that enables specification and verification of forwarding policies. Using this approach, a precise and unambiguous meaning is given to a forwarding policy specification, and it is possible to automatically detect anomalies in forwarding policy specifications (e.g., due to human errors made by network administrators, service providers, tenants, etc.).

Note that we just consider errors in forwarding policies and we do not address the possible errors in the translation algorithm that generates OpenFlow flow entries from policies. Since policy translation is performed automatically, we rely on the correctness of this process and leave its verification out of scope.

Differently from our proposal, which focuses on checking forwarding policies themselves, so far literature has mainly addressed the verification of OpenFlow flow entries, thus working on the output of the forwarding policy translation process rather than on its input. More precisely, the existing proposals (e.g., VeriFlow [7] and FlowChecker [8]) mainly focus on detecting the presence of conflicts among flow entries, which consists in checking the violation of some network invariants or checking when a traffic flow is enabled (or disabled) by a new flow entry, while the previous FlowTable entries disabled (or enabled) that flow.

One key factor that distinguishes our work from these approaches is *early detection* of forwarding policies, which means performing a verification step *before* translating such policies into OpenFlow entries. The verification process has to be executed before the Controller configures the network, i.e providers will run the verification either within or on a layer above the Controller. Early detection can speed up the fixing of problems. This verification, in fact, can already be performed during the policy specification phase. Early-detection of a number of anomalies allows one to immediately fix the detected anomalies, without even starting the deployment phase. In this way it is possible also to avoid the waste of computational resources spent for translating anomalous forwarding policies.

Of course, the translation tools based on the policy-oriented specification languages already mentioned [5, 6] perform some integrity check before translating a policy specification into OpenFlow entries, but these languages miss a formal underlying model and these preliminary checks have not been the object of publications so far.

One new contribution of our work is thus the definition of a formal model that allows the automatic detection of a set of anomalies in forwarding policies. By *anomaly* we mean an erroneous specification of forwarding policies, which may cause misleading network conditions and states[1]. For example, anomalies can be related to violation of administrator-defined requirements, but also conflicting forwarding specifications. An example of an administrator-defined constraint can be related to the network function ordering within a service chain, while an example of conflicting forwarding specification is when two policy rules specify different chains to be traversed by the same packet flow. These rules could lead to the generation and installation of conflictual OpenFlow flow entries in network switches.

Furthermore, the proposed model supports high flexibility in defining the anomalies to be checked. This is achieved by defining a set of operators that let one precisely and unambiguously specify the anomalies to be checked. The supported operators have been in part inspired by the previous works on conflict analysis performed in

[1] Errors in network forwarding may be still present after the early-detection of anomalies in forwarding policies, due to wrong configuration installed into the network functions involved in the network (e.g., wrong filtering rules installed in firewalls). To detect and solve this kind of errors, an administrator needs to use other approaches, like the other verification approach presented in this thesis (Chapter 3).

other policy domains (e.g., OpenFlow [9] and traffic filtering [10]). The anomalies specified by means of these operators are automatically translated into formulas in First Order Logic (FOL) that are finally fed to the verifier along with the policy to be checked. The flexibility introduced by this approach allows administrators to define their custom anomalies but, at the same time, it is possible to create a set of pre-defined anomalies, specified using the same formalism, corresponding to general mistakes to be avoided in any network.

We also propose a hierarchy of classes of anomalies that can arise in a forwarding policy. Such hierarchy considers both those anomalies that may lead to the OpenFlow conflicts already presented in literature, and it includes also new classes of anomalies, proper of the forwarding policy domain, that can be detected thanks to our model.

The rest of the chapter is organized as follows: the proposed model is presented in details in Section 2.2 where we describe the structure of a forwarding policy and the supported operators for describing anomalies; Section 2.3 presents the main classes of anomalies the model allows one to detect. We have also implemented an anomaly detection process, in order to evaluate the time required to verify a network policy (Section 2.4). Finally, Sections 2.5 concludes the chapter presenting the current state of the art.

2.2 The approach

In this section, we present the formal model proposed for verifying forwarding policies. First, we present the model of a forwarding policy, which is a set of rules (namely *forwarding rules* or simply *rules*) that manage several traffic flows. Then, we present the comparison operators that can be used for building anomaly specifications. Such operators enable pairwise comparisons between the elements that compose a forwarding rule or that belong to different rules. Finally we introduce the anomaly model, which is a FOL formula that involves a set of pairwise comparisons. In the next section we identify a possible classification of anomalies, in order to better clarify the new classes of anomalies detected thanks to the proposed model. We also describe how the anomalies of the different classes are specified and how their formulas are built.

2.2.1 Forwarding policy model

In our model, the flow management of a network is specified through a forwarding policy ($\mathbb{R}_{\mathbb{F}}$). A policy is a set of forwarding rules, each one putting in relation traffic flows with the SFCs those flows can traverse at run-time. A generic forwarding rule r in a forwarding policy ($r \in \mathbb{R}_{\mathbb{F}}$) has the following structure:

$$r = (\mathscr{M}, \mathscr{C}),\ r \in \mathbb{R}_{\mathbb{F}} \tag{2.1}$$

where:

(i) $\mathscr{M}$ is the traffic flow managed by the rule, which belongs to the set of all possible flows in a network ($\mathscr{M} \subseteq \mathbb{M}$);

(ii) $\mathscr{C}$ is the set of SFCs that $\mathscr{M}$ can potentially traverse at run-time and it is part of the whole set of chains instantiated in the network ($\mathscr{C} \subseteq \mathbb{C}$).

In our model we suppose that a flow $\mathscr{M}$ does not traverse necessarily all the configured chains at run-time. In a real network scenario, packet forwarding, in fact, depends also on function configuration and state, thus a flow can be forwarded to zero, one, many or all of the allowed chains. An example is when web traffic traverses a load-balancer, which selects only one outgoing path, based on its internal algorithm and state. Another example is a mirroring function in which case the same traffic flow follows different chains.

The proposed model does not consider rule priorities as instead it has been done in the OpenFlow domain ([9], [11]). This is because we are working at a higher abstraction level, where we loose the notion of order among forwarding rules. It is only when a forwarding policy is translated into OpenFlow flow entries that we need a priority in FlowTables. Another reason for omitting priorities is also that forwarding rules should be specified to manage non-overlapped traffic flows. When this condition is violated, we have an anomaly in the policy according to our model.

A flow $\mathscr{M}$ is modelled by referring to a set of OpenFlow fields $\mathscr{N}$, named *network fields*. In detail, a network field n is an element of $\mathscr{N}$ ($n \in \mathscr{N}$) and $\mathscr{N}$ is currently defined as follows[2]:

$$\begin{aligned}\mathscr{N} = \{&eth\_src, eth\_dst, eth\_type, vlan\_id,\\ &ip\_src, ip\_dst, ip\_proto, port\_src, port\_dst\}\end{aligned} \tag{2.2}$$

Each network field has a type, i.e. the set of values that can be taken by the field. If the type of a network field n is a totally ordered set, when defining a flow $\mathscr{M}$, it is possible to specify that $\mathscr{M}$ includes the packets for which n takes either a single value or a range of values, based on the granularity we want to use in specifying $\mathscr{M}$. For example, in a flow specification we can use $ip\_dst = 8.8.8.0/24$ or $port\_dst = [80, 100]$, because the types of IP address and port number fields are ordered sets of values, but if we prefer we can also use single values (e.g., $ip\_dst = 8.8.8.151$ or $port\_dst = 80$).

In general, in order to identify a particular flow, a value (or range of values) v has to be specified for each supported network field n. The packets that belong to the flow are those whose network fields match the specified values. As a special case, it is possible to use the special value * for a network field n, which means that n can take any value according to its type. Hence, a flow $\mathscr{M}$ is formally defined by a list of equalities, one for each network field:

$$\begin{aligned}\mathscr{M} = (&eth\_src = v_{eth\_src},\ eth\_dst = v_{eth\_dst},\ eth\_type = v_{eth\_type},\\ &vlan\_id = v_{vlan\_id},\ ip\_src = v_{ip\_src},\ ip\_dst = v_{ip\_dst},\\ &ip\_proto = v_{ip\_proto},\ port\_src = v_{port\_src},\ port\_dst = v_{port\_dst})\end{aligned} \tag{2.3}$$

As specified in (2.1), a forwarding rule also includes the *service chains* (i.e., SFCs) $\mathscr{C}$ that can be traversed by the flow $\mathscr{M}$. In detail, $\mathscr{C}$ is the set of chains c enforced by a rule, which is, in turn, a sub-set of all possible chains $\mathbb{C}$:

$$\mathscr{C} = \{c^1, c^2 ..., c^n\}, \quad c^k \in \mathscr{C} \subseteq \mathbb{C} \tag{2.4}$$

Each chain $c^k \in \mathscr{C}$ is represented in our model as an ordered sequence of network functions $c^k = [f^{1_k}, f^{2_k}, ..., f^{m_k}]$. Each function f^{w_k} in a chain c^k is one of the

[2] This set of fields can be extended as needed.

functions present in the network and it is modelled by the pair:

$$f^{w_k} =< f\_id^{w_k}, f\_type^{w_k} > \tag{2.5}$$

where $f\_id^{w_k}$ is the function identifier and $f\_type^{w_k}$ is the type of the function, which necessarily has to belong to the VNF catalogue ($\mathbb{F}$) offered by the operator[3]. Thus we model a network chain as:

$$c^k =[< f\_id^{1_k}, f\_type^{1_k} >,...,< f\_id^{m_k}, f\_type^{m_k} >], f\_type^{w_k} \in \mathbb{F} \tag{2.6}$$

To summarize, a forwarding rule r is modelled as follows:

$$\begin{aligned} r = (\mathscr{M},\mathscr{C}) = (\ & (eth\_src = v_{eth\_src},\ eth\_dst = v_{eth\_dst},\\ & eth\_type = v_{eth\_type}, vlan\_id = v_{vlan\_id},\ ip\_src = v_{ip\_src},\\ & ip\_dst = v_{ip\_dst},\ ip\_proto = v_{ip\_proto},\ port\_src = v_{port\_src},\\ & port\_dst = v_{port\_dst}),\ \{[< f\_id^{1_1}, f\_type^{1_1} >,...]\ ,...\ \}\ ) \end{aligned} \tag{2.7}$$

As an example of forwarding rule, let us consider to have to manage the web traffic from a client with address 130.192.225.116 to a web server in the network scenario depicted in Figure 2.1. We can specify this traffic can traverse two possible chains with the following forwarding rule:

$$\begin{aligned} r = ((& eth\_src = *,\ eth\_dst = *,\ eth\_type = 0\mathbf{x}0800,\\ & vlan\_id = *,\ ip\_src = 130.192.225.116,\ ip\_dst = 8.8.8.0/24,\\ & ip\_proto = 0\mathbf{x}06,\ port\_src = *,\ port\_dst = 80),\\ & \{[< host\_a, H >, < lb\_a, LB >, < ids\_a, IDS >, < fw, FW >,\\ & < lb\_b, LB >, < server\_a, S >], [< host\_a, H >, < lb\_a, LB >,\\ & < ids\_b, IDS >, < fw, FW >, < lb\_b, LB >, < server\_a, S >]\}) \end{aligned}$$

In the proposed formalism, we explicitly indicate hosts in the SFC specification. This is because we consider the hosts as part of the service chain. However, many

[3] From now on, we use abbreviations in the formulas to indicate the type of network function involved in the network chains. In particular, we use these abbreviations in the next examples: H (End Host), WS (Web Server), FW (firewall), NAT (network address translator), DPI (deep packet inspection), MN (monitor), LB (load-balancer), SPAM (anti-spam), CACHE (web-cache), IDS (intrusion detection system), VPN (virtual private network) and L7_FW (layer 7 firewall).

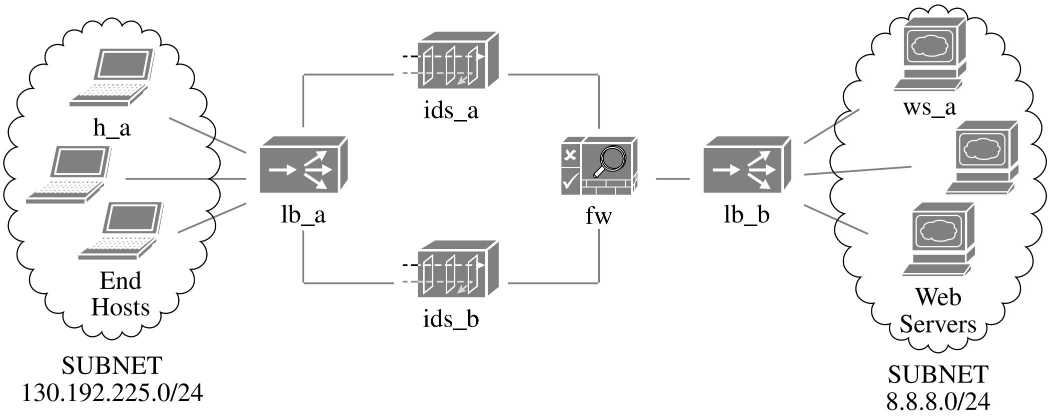

Fig. 2.1 Topology example.

formalisms to define SFCs in different manners are possible, for example by including host identification in the traffic specification ($\mathscr{M}$). Thus, we leave this aspect and its effects on the verification approach as future improvements of our model.

From now on, we indicate the elements of a forwarding rule r_i as follows:

(i) $\mathscr{M}_i$ and $\mathscr{C}_i$ are respectively the flow and the SFCs managed in rule r_i;

(ii) n_i^h is the h-th network field in r_i and v_i^h is its value specification;

(iii) c_i^k specifies the k-th chain in the i-th rule;

(iv) $f_i^{w_k}$ is the w-th function in c_i^k.

We use this notation in case we are referring to different forwarding rules (e.g., r_i and r_j), while in case we are indicating a single rule, we do not use any index as subscript to indicate the rule itself r and its elements ($\mathscr{M}$, $\mathscr{C}$, and their network fields, values and chains).

2.2.2 Comparison operators for anomaly specification

In order to enable the specification of anomalies, the model offers a set of relational operators. These operators enable the specification of pairwise comparisons ($x \in \mathscr{X}$), each one involving network fields and SFCs belonging to the same or to different rules. Formally, these comparisons are predicates that let us finally identify sets of matching forwarding rules. More precisely, if x is a comparison that involves fields and SFCs belonging to the same generic rule r, x can be regarded as a function of r which returns the result (true or false) of the comparison evaluated on r. Moreover, x identifies the set of rules r such that $x(r)$ is true. If instead x involves fields and SFCs belonging to two different rules r_i, r_j, x can be regarded as a function of two variables r_i, r_j which returns the result (true or false) of the comparison evaluated on r_i and r_j. Moreover, x in this case identifies the set of pairs of rules (r_i, r_j) such that $x(r_i, r_j)$ is true.

The set of operators offered in this model was inspired by the previous works presented in literature and in particular, we have considered two proposals of conflict analysis: one regarding firewall rules (i.e., a filtering policy domain [10]) and the second one in the context of OpenFlow rules [9].

Since those proposals are applied to different policy domains, we have adapted the sets of operators proposed in those works in order to enable the verification of forwarding rules. In particular, we exploited the existing network field-related operators to establish the relationships between traffic flows. Moreover, we introduced SFC comparisons, and other newly defined operators in order to enable the specification of a wider set of anomalies.

Network field operators

The set of supported operators to compare network fields includes:

- **equivalence** ($=$): two fields are equivalent if the value(s) they can take (in the flows of the rules they belong to) are the same, even though they are different fields (e.g., if $\mathscr{M}$ includes $port\_src = 80$ and $port\_dst = 80$, then, for rule r, which includes $\mathscr{M}$, we have $port\_src = port\_dst$). In case the fields are specified to take ranges of values in a flow, they are equal if the set of values they can take in their flow is the same. Of course, equivalence can also be applied to fields belonging to the flows of different rules;
- **majority** ($>$): this operator can be applied to fields that are specified to take single numeric values. In this case, a network field is grater than another one or than a specific value, when their values have this relation (e.g., if $port\_src = 70001$, $port\_src > 65535$ is true);

- **dominance** ($\succ$): this operator can be applied to fields that can take ranges of values. In particular, a field dominates another one when it can take all the values that can be taken by the second field (e.g., if $port\_src_i = [1024, 2048]$ and $port\_src_j = [1024, 1500]$, then $port\_src_i \succ port\_src_j$);
- **correlation** ($\sim$): this operator can be applied to fields that can take ranges of values. Two fields are correlated if they share some values but none dominates the other (e.g., if $port\_src_i = [1024, 1500]$ and $port\_src_j = [0, 1100]$, then $port\_src_i \sim port\_src_j$).
- **disjointness** ($\perp$): two network fields are disjoint if they do not share any value (e.g., if $port\_src_i = [1000, 1024]$ and $port\_src_j = [1100, 8080]$, then $port\_src_i \perp port\_src_j$).

The model also offers the negative form of the aforementioned operators, like *non-disjointness* ($\not\perp$ - two fields are either equal, correlated or one dominates the other) and *non-equivalence* ($\neq$ - two fields can be correlated, disjoint or one can dominate the other).

Moreover, combinations of operators are allowed. This is the case of *equivalence or dominance* ($\succeq$ - i.e., a network field is equivalent to or dominates another one), *equivalence or correlation* ($\simeq$ - i.e., a network field is equivalent or correlated to another one) and *equivalence or majority* ($\geq$ - i.e., a network field is equivalent or grater than another field).

SFC operators

As already mentioned, here we define new operators, in order to enable comparisons that involve SFCs. First, we introduce the following notation to represent ordered sequences (i.e., SFCs) and unordered sets of network functions:

- **ordered sequence** ([]): this notation was already introduced for the specification of SFCs inside forwarding rules. It is also used to represent ordered sequences of network functions in an anomaly specification. Via the wildcard character *, the proposed model supports the specification of unidentified functions, i.e. functions for which only the type is specified, not the identity. For example, a chain composed of a NAT followed by a firewall can be specified generically as $[< *, NAT >, < *, FW >]$;

- **set** ($\{\}$): this notation can be used to specify unordered collections of functions. For example, a chain including an application firewall and a DPI, non necessarily in this order, can be specified as $\{< *, L7\_FW >, < *, DPI >\}$.

For what concerns the comparison between SFCs that can belong to the same forwarding rule or to different rules, we extend the current literature by enabling pairwise comparisons between: *(i)* two chains of either the same or different rules; *(ii)* a chain and an ordered sequence of functions (i.e., a chain not managed by a forwarding rule); *(iii)* chain and a set of functions. In some cases, the same operators can be used for different types of comparisons, the exact meaning of the comparison being determined by the types of the compared elements. In case of comparison between two chains (of the same or of different rules - e.g., c^k and c^l) or a chain and an ordered sequence (e.g., c^k and $[f^1, f^2, ..]$), the following operators can be used:

- **equivalence** ($=$): two chains are equivalent if they are the same ordered sequence of network functions (e.g., if $c^k = [f^1, f^2]$ and $c^l = [f^1, f^2]$, then $c^k = c^l$);
- **dominance** ($\succ$): a chain dominates another one when it contains the second chain as a subsequence and the two chains are not equivalent (e.g., if $c^k = [f^1, f^2, f^3]$ and $c^l = [f^2, f^3]$, then $c^k \succ c^l$);
- **correlation** ($\sim$): two chains are correlated if none dominates the other, but they share an ordered sub-chain (e.g., if $c^k = [f^1, f^2, f^3]$ and $c^l = [f^4, f^2, f^3]$, then $c^k \ \sim c^l$);
- **disjointness** ($\perp$): two chains are disjoint if they do not have any sub-chain in common (e.g., if $c^k = [f^1, f^2, f^3]$ and $c^l = [f^4]$, then $c^k \perp c^l$).

The comparison between a chain and an unordered set of functions (i.e., c^k and $\{f^1, f^2, ...\}$), instead, can involve the following operators:

- **correlation** ($\sim$): a chain is correlated to a set of functions if it contains some of those functions (e.g., $c^k = [f^1, f^2, f^3] \sim \{f^4, f^2, f^3\}$);
- **disjointness** ($\perp$): a chain and a set of functions are disjoint if they do not share any function (e.g., $c^k = [< spam, SPAM >, < fw, FW >, < dpi, DPI >] \perp \{< mn, MN >\}$);

- **inclusion** ($\subset$): a set of non-ordered network functions is included into a chain if all of its functions are part of the chain (e.g., if $c^k = [< spam, SPAM >, < nat, NAT >, < fw, FW >]$, then $\{< nat, NAT >\} \subset c^k$).

It is interesting to note that in some particular cases, the inclusion and dominance operators take the same meaning. Let us consider, for example, that one wants to specify the condition that a network function f belongs to a chain c. This condition can be expressed either by the comparison $\{f\} \subset c$ or by $c \succ [f]$. However, it is better to have different operators, in order to cover a richer set of anomalies and keep expressions as simple as possible. For example, if the model would support only the dominance operator, an administrator could specify that m functions f^1,f^2,..., f^m, not necessarily in this order, belong to a SFC, by specifying

$$c \succ [f^1] \wedge c \succ [f^2] \wedge .... \wedge c \succ [f^m]$$

Supporting also the inclusion operator ($\subset$), we then make the formula syntax less complex and less likely to be mistaken:

$$\{f^1, f^2, ..., f^m\} \subset c$$

For SFC comparisons too, the model offers the negative forms of the aforementioned operators (e.g., *non-correlation* ($\not\sim$), *non-dominance* ($\not\succ$), etc.), and some combinations of operators (i.e., *equivalence or dominance* ($\succeq$), *equivalence or correlation* ($\simeq$) and *inclusion or equivalence* ($\subseteq$)). Note that the meaning of these additional operators changes based on the type of the operands, as specified previously.

In order to further enlarge the expressive power, we define also another new operator that lets us specify comparisons related to the position of a function within a service chain:

- $\pi(f, c)$ returns the position of network function f within chain c, if $\{f\} \subseteq c$, or 0 otherwise. Let us consider for example $c = [< nat, NAT >, < fw, FW >]$. The NAT position inside c is $\pi(< nat, NAT >, c) = 1$.

Finally, the model also allows to check the membership of a chain c^k within a set of chains $\mathscr{C}_i$:

- **membership** ($\in$): this boolean operator returns true if a chain c^k belongs to the set of chains $\mathscr{C}_i$ of the forwarding rule r_i and false otherwise. For example, let us consider $c^k = [< spam, SPAM >, < fw, FW >, < dpi, DPI >]$ and $\mathscr{C} = \{[< nat, NAT >, < fw, FW >], [< spam, SPAM >, < fw, FW >, < dpi, DPI >]\}$, then $c \in \mathscr{C}$ is true. The model supports also the negative form of this operator (i.e, $\notin$).

2.2.3 Anomaly model

Forwarding anomalies, or simply *anomalies*, represent erroneous and undesired network conditions that a network administrator wants to detect and eliminate in a forwarding policy, in order to guarantee a correct traffic forwarding. The set of all anomalies is denoted $\mathbb{A}$.

More precisely, an anomaly $a \in \mathbb{A}$ represents a single network condition that an administrator wants to avoid. Formally, it is a predicate defined on one or more rules. For example, if r is a variable that represents a rule, an anomaly can be formally represented by a function $a(r)$ that returns the boolean true if the anomaly is present in the single rule r and false otherwise. Similarly, if r_i and r_j are two rules, an anomaly can be defined as a function $a(r_i, r_j)$ that returns the boolean true if the anomaly is present in the pair of rules (r_i, r_j). A policy $\mathbb{R}_\mathbb{F}$ is anomaly-free if $\forall a \in \mathbb{A}$ we have $a(r) = false\ \forall r \in \mathbb{R}_\mathbb{F}$ or $a(r_i, r_j) = false\ \forall (r_i, r_j) \in \mathbb{R}_\mathbb{F} \times \mathbb{R}_\mathbb{F}$, according to the arity of a.

In detail, an anomaly is formally specified by a set of Horn clauses [12] that involve pairwise comparisons. Each Horn clause is a conjunction of positive comparisons x_i on rule fields and chains, which implies the presence of the anomaly in a single rule or in a pair of rules. Hence, the structure of Horn clauses that define anomalies is as follows:

$$\begin{aligned} x_1 \land x_2 \land \ldots \land x_q \to a(r), &\quad a \in \mathbb{A} \\ x_1 \land x_2 \land \ldots \land x_q \to a(r_i, r_j), &\quad a \in \mathbb{A} \end{aligned} \tag{2.8}$$

In practice, the intersection of the sets of rules identified by the comparisons that occur in the left hand side of the formula is the set of rules in which the anomaly is present.

In order to be flexible enough, the model supports also existential ($\exists$) and universal ($\forall$) quantification over SFCs in the left hand side of the Horn clauses, which enables the possibility to specify that some comparisons have to be satisfied by at least one or by all the SFCs of a forwarding rule. In this model, when we quantify universally on pairs of chains, we are considering implicitly pairs of different chains. For example, in case we check the correlation among the SFCs in a forwarding rule, we can specify $c_i^k \sim c_i^l$, $\forall c_i^k, c_i^l \in \mathscr{C}_i$ to check only pairs of different chains, without indicating explicitly that $k \neq l$.

An example of anomaly that refers to pairs of rules and that uses universal quantification is the rule duplication anomaly, which occurs when a policy includes two identical rules. This anomaly can be specified as the anomaly that is true when the pairwise equivalence between all the elements of two rules (r_i and r_j), including all the SFCs, is satisfied:

$$\begin{aligned}
&eth\_src_i = eth\_src_j \ \wedge \ eth\_dst_i = eth\_dst_j \ \wedge \ eth\_type_i = eth\_type_j \ \wedge \\
&vlan\_id_i = vlan\_id_j \ \wedge \ ip\_src_i = ip\_src_j \ \wedge \ ip\_dst_i = ip\_dst_j \ \wedge \\
&ip\_proto_i = ip\_proto_j \ \wedge port\_src_i = port\_src_j \ \wedge \ port\_dst_i = port\_dst_j \ \wedge \\
&c_i^k \in \mathscr{C}_j, \forall c_i^k \in \mathscr{C}_i \ \wedge \ c_j^l \in \mathscr{C}_i, \forall c_j^l \in \mathscr{C}_j \rightarrow \ Duplication(r_i, r_j)
\end{aligned} \tag{2.9}$$

Another example of anomaly can be defined with reference to Figure 2.1. A network administrator who wants to make sure all web traffic traverses a firewall can define a custom anomaly triggered if web traffic may not traverse a firewall. This anomaly, which involves a single rule, can be expressed by means of the following formula:

$$\begin{aligned}
&eth\_src = * \ \wedge \ eth\_dst = * \ \wedge \ eth\_type = 0\mathbf{x}0800 \ \wedge \ vlan\_id = * \ \wedge \\
&ip\_src = 130.192.225.116 \ \wedge \ ip\_dst = 8.8.8.0/24 \ \wedge \ ip\_proto = 0\mathbf{x}06 \wedge \\
&port\_src = * \ \wedge \ port\_dst = 80 \ \wedge \ \{< *, FW >\} \not\subset c^k \ , \\
&\forall c^k \in \mathscr{C} \rightarrow \ webNoFirewall(r)
\end{aligned} \tag{2.10}$$

In the next section, we present a proposal of anomaly classification, which highlights the hierarchy of anomalies the model can express. For each class identified (depicted also in Figure 2.2), an anomaly takes a specific form, based on the comparisons x used to express the anomaly.

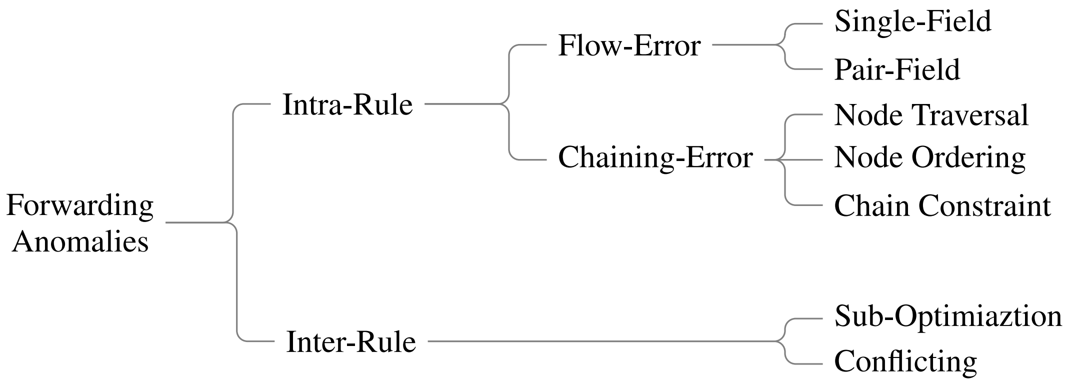

Fig. 2.2 Hierarchy of anomaly classes.

2.3 Anomalies Classification

Forwarding anomalies are divided into several classes, based on the kind of pairwise comparisons x involved in the Horn clauses that express them.

The classification we are proposing is based mostly on the object of comparison rather than on the comparison operator occurring in the formula. For this reason, in the following formulas that we use for defining the classification we leave the operator unspecified and we indicate it generically by the $\star$ symbol. In other words, the proposed classification distinguishes anomalies according to the kind of operands of comparisons. In particular, we distinguish anomalies whose comparison operators involve:

(i) a network field and a specific value ($n \star v$);

(ii) network fields either in the same ($n^h \star n^g$) or in different rules ($n_i^h \star n_j^g$);

(iii) chains of the same forwarding rule ($c^k \star c^l$);

(iv) a chain and a sequence ($c \star [f^1, f^2, ..., f^m]$) or a chain and an unordered set of functions ($c \star \{f^1, f^2, ..., f^m\}$);

Figure 2.2 shows the proposed anomaly classification. Note that the leaf anomalies of the hierarchy shown in Figure 2.2 are not exhaustive, but they just cover the most important cases. More classes can be added to the hierarchy. Firstly the hierarchy splits anomalies into two macro-classes:

(i) Intra-Rule anomalies, arising within a single forwarding rule and involving errors on network field or chain specifications;

(ii) Inter-Rule anomalies, involving pairwise comparisons on network fields or SFCs of different forwarding rules.

Intra-Rule anomalies are further divided into two sub-classes, concerning the flow- or chain-related parts of the forwarding rule model (i.e., the Flow and Chaining classes). Note that the leaf anomalies of the hierarchy shown in Figure 2.2 are not exhaustive, but they just cover the most important cases. More classes can be added to the hierarchy.

The *Flow-Error* class contains the anomalies that have been identified in literature [11] as the errors derived from a bad definition of the flow $\mathscr{M}$. This first class includes two sub-classes:

(i) *Single-Field* anomalies check the correctness of single network fields in a forwarding rule;

(ii) *Pair-Field* anomalies verify the relationships among different network fields within a forwarding rule.

The *Chaining* class includes the new types of anomalies specific for the forwarding policy domain. The anomalies of this class, in fact, can be specified thanks to the new set of chain-related operators, defined in Section 2.2.2.

The Chaining class is further divided into three sub-classes:

(i) *Node Traversal* includes the violations of requirements related to the traversal of a set of VNFs by a traffic flow;

(ii) *Node Ordering* anomalies are related to the violation of ordering constraints of a set of VNFs that makes up a chain;

(iii) *Chain Constraint* anomalies occur when the set of chains traversed by a traffic flow violates some administrator-defined invariants.

Finally, the proposed anomaly classification includes Inter-Rule anomalies. The literature locates under this class two types of anomalies:

(i) *Sub-Optimization* anomalies correspond to conditions in which the forwarding rules will be translated into flow entries that under-optimize the available resources in network nodes (e.g., memory consumption in switches);

(ii) *Conflicting* anomalies correspond to the presence of rules that would lead to the creation of conflictual flow entries.

The leaf classes of the hierarchy are described in more detail in the next subsections. This classification could be further refined by introducing additional classes. Note, in fact, that the proposed classification does not include those anomalies that are combinations of the presented classes. Of course, the model is able to treat and detect also this kind of anomalies. However, we preferred to do not consider this further classes in order to encourage administrators to build simple anomalies. The specification of complex anomalies can bring administrators to overlook some possible cases of anomaly and leave undetected the corresponding forwarding rules. The use of simple anomalies will thus imply that a forwarding rule may trigger even more than one anomaly.

Single-Field anomalies

The anomalies in this class are those that involve only comparisons between single network fields and specific values. Thus, following the generic anomaly structure defined in (2.8), a comparison x that composes a single field anomaly is expressed as:

$$x = n \star v \tag{2.11}$$

We recall also that n and v are respectively a generic network field and a generic value (or range of values) it can take in a forwarding rule r, while $\star$ stands for one of the operators defined in Section 2.2.2.

Examples of anomalies of this class are the ones triggered when port numbers are grater than their maximum values. Such kind of anomalies belong to the set of pre-defined anomalies, since they are mistaken policy specifications in every network topology. They can be expressed by the following formulas:

$$port\_src > 65535 \rightarrow BadPortSrc(r) \tag{2.12}$$

$$port\_dst > 65535 \rightarrow BadPortDst(r) \tag{2.13}$$

Field-Pair anomalies

Another sub-class of Flow anomalies is the Field-Pair class. A Field-Pair anomaly is one that contains only pairwise comparisons of different network fields belonging to the same forwarding rule, like source and destination IP. Thus a generic Field-Pair anomaly is one expressed by means of comparisons that take the following form:

$$x = n^h \star n^g \tag{2.14}$$

An example of Field-Pair anomaly is when source and destination IP addresses are the same, which is specified by means of the following formula:

$$ip\_src = ip\_dst \rightarrow BadIpAddress(r) \tag{2.15}$$

Of course, this is another example of anomaly included in the pre-defined set of anomalies supported by our model.

Node Traversal anomalies

These anomalies are those that arise when a traffic flow can (or cannot) traverse one or more network functions. Hence, such anomalies are expressed by comparing network fields with specific values and chains with ordered or non-ordered sets of functions. The forms of comparisons in these anomalies are:

$$\begin{aligned} & x = n \star v \\ & x = c \star [f^1, f^2, ..., f^m] \\ & x = c \star \{f^1, f^2, ..., f^m\} \end{aligned}$$

where c is a generic chain specifier which can be existentially or universally quantified. In order to identify when a custom anomaly like “`Web traffic does not pass through an IDS`” is triggered in a case scenario like the network shown in Figure 2.1, we can use the following anomaly belonging to this class:

$$eth\_src = * \wedge eth\_dst = * \wedge eth\_type = 0\mathbf{x}0800 \wedge$$

$$
\begin{aligned}
&vlan\_id = * \wedge ip\_src = 130.192.225.0/24 \wedge \\
&ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge \\
&port\_src = * \wedge port\_dst = 80 \wedge \\
&c^k \not\succ [< *, IDS >], \forall c^k \in \mathscr{C} \rightarrow NoWeb2IDS(r)
\end{aligned}
$$

Node Ordering anomalies

This class contains anomalies that are violations of ordering constraints on the functions traversed by a flow. Such constraints can be expressed in terms of the position of the w-th network function within a chain c (i.e., $\pi(f^w, c)$) and they may be required to hold for at least one or for all the chains of the forwarding rules that manage that flow. Of course, in order to express the flow for which the constraint is checked, network field comparisons can be used. Hence these anomalies are specified by formulas including the following comparisons:

$$
\begin{aligned}
&x = n \star v \\
&x = \pi(f^w, c) \star \pi(f^q, c) \quad \text{with } f^w, f^q \in c
\end{aligned}
$$

An example is when we want to ensure that a NAT is always configured to process traffic before a firewall. This means that we have to detect the anomalous situation when a NAT is located after a firewall in the SFC topology, which can be done by the following anomaly definition based on the position operator:

$$
\begin{aligned}
&eth\_src =^* \wedge eth\_dst =^* \wedge eth\_type =^* \wedge vlan\_id =^* \wedge \\
&ip\_src =^* \wedge ip\_dst =^* \wedge ip\_proto =^* \wedge port\_src =^* \wedge \\
&port\_dst =^* \wedge \pi(< *, NAT >, c) > \pi(< *, FW >, c), \\
&\exists c \in \mathscr{C} \rightarrow NatAfterFW(r)
\end{aligned}
\tag{2.16}
$$

Chain Constraint anomalies

This category includes anomalies aimed at detecting conditions that apply to all the chains defined where the anomaly is triggered by comparing such chains between each other (e.g., all the chains in a forwarding rule are equal). Thus such anomalies contain comparisons between SFCs ($c^k \star c^w$) that belong to the same forwarding rule and network fields comparisons to identify the flow ($n \star v$ or $n^h \star n^g$):

$$x = n \star v$$
$$x = n^h \star n^g$$
$$x = c^k \star c^l$$

Let us consider that a web traffic is balanced on two chains (Figure 2.1) and it must be processed either by the same (i.e., equivalent chains) or by a similar set of network functions (i.e., correlated or dominated chains) in the two chains. This means that the two chains must not be disjoint and we can detect this anomalous situation by means of the following anomaly definition:

$$\begin{aligned}
&eth\_src = * \wedge eth\_dst = * \wedge eth\_type = 0\mathbf{x}0800 \wedge \\
&vlan\_id = * \wedge ip\_src = 130.192.225.116 \wedge ip\_dst = 8.8.8.0/24 \wedge \\
&ip\_proto = 0\mathbf{x}06 \wedge port\_src = * \wedge port\_dst = 80 \wedge \\
&c^k \perp c^l, \forall c^k, c^l \in \mathscr{C} \rightarrow DisjointChains(r)
\end{aligned}$$

Sub-Optimization anomalies

Such anomalies detect if more forwarding rules enforce the same set of SFCs. Hence, we locate under this category those anomalies that aim at detecting under-optimizations of the policy specification and thus situations where more forwarding rules can be substituted by a single rule. In order to detect such anomalies, it is necessary to discover the forwarding rules that have the same sets of chains. Hence the anomalies in this class include the following comparisons:

$$x = n_i^h \star n_j^g$$
$$x = c_i^k \in \mathscr{C}_j, \forall c_i^k \in \mathscr{C}_i \wedge c_j^l \in \mathscr{C}_i, \forall c_j^l \in \mathscr{C}_j$$

Under this class, we include the duplication anomaly defined in (2.9). Another example is the following anomaly, where we detect those forwarding rules that refer to completely disjoint traffic flows but that enforce the same set of SFCs:

$$eth\_src_i \neq eth\_src_j \wedge eth\_dst_i \neq eth\_src_j \wedge eth\_type_i \neq eth\_type_j \wedge$$

$$\begin{aligned}
&vlan\_id_i \neq vlan\_id_j \ \wedge \ ip\_src_i \perp ip\_src_j \ \wedge \ ip\_dst_i \perp ip\_dst_j \ \wedge \\
&ip\_proto_i \neq ip\_proto_j \ \wedge \ port\_src_i \perp port\_src_j \ \wedge \ port\_dst_i \perp port\_dst_j \ \wedge \\
&c_i^k \in \mathscr{C}_j, \forall c_i^k \in \mathscr{C}_i \ \wedge \ c_j^l \in \mathscr{C}_i, \ \forall c_j^l \in \mathscr{C}_j \rightarrow \ SubOptimizedFlows(r_i, r_j)
\end{aligned}$$

Conflicting anomalies

In our model, conflicts arise when two forwarding rules manage the same traffic flow but they do not specify the same sets of chains. If the two rules are installed into the network, inconsistencies in the traffic forwarding can be generated at run-time. Hence the formula for detecting this kind of anomaly includes the following comparisons:

$$\begin{aligned}
&x = n_i^h \star n_j^g \\
&x = c_i^k \in \mathscr{C}_j, \forall c_i^k \in \mathscr{C}_i \\
&x = c_i^k \notin \mathscr{C}_j, \forall c_i^k \in \mathscr{C}_i \\
&x = c_i^k \in \mathscr{C}_j, \exists c_i^k \in \mathscr{C}_i \\
&x = c_i^k \notin \mathscr{C}_j, \exists c_i^k \in \mathscr{C}_i
\end{aligned} \tag{2.17}$$

The comparisons x that compose a conflicting anomaly have been selected so as to enable the specification of different types of relationships between two sets of chains. An example of relationship is the case in which two sets $\mathscr{C}_i$ and $\mathscr{C}_j$ contain the same SFCs or also when C_i contains all the chains of C_j as subset. Another case is when C_i and C_j do not have any chain in common. This means that the policy contains two forwarding rules that forward the same traffic flow to different sets of SFCs. This kind of conflictual anomaly can be detected by the following formula:

$$\begin{aligned}
&eth\_src_i = eth\_src_j \ \wedge \ eth\_dst_i = eth\_dst_j \ \wedge \\
&eth\_type_i = eth\_type_j \ \wedge \ vlan\_id_i = vlan\_id_j \ \wedge \\
&eth\_type_i = eth\_type_j \ \wedge \ vlan\_id_i = vlan\_id_j \ \wedge \\
&ip\_src_i = ip\_src_j \ \wedge \ ip\_dst_i = ip\_dst_j \ \wedge \\
&ip\_proto_i = ip\_proto_j \ \wedge port\_src_i = port\_src_j \ \wedge \\
&port\_dst_i = port\_dst_j \ \wedge \ c_i^k \notin \mathscr{C}_j, \ \forall c_i^k \in \mathscr{C}_i \ \wedge \\
&c_j^l \notin \mathscr{C}_i, \ \forall c_j^l \in \mathscr{C}_j \rightarrow \ DisjointChains(r_i, r_j)
\end{aligned} \tag{2.18}$$

Note that some cases of "conflicting" forwarding rules according to the anomaly model (2.17) may not be considered by the administrators as conflicting anomalies. This is because this kind of anomalies depends on the network topology and on what the administrator considers erroneous for her network. Let us consider the case of two forwarding rules r_i and r_j to forward the traffic between the end-host *h_a* and the web server *ws_a*:

$$\begin{aligned} r_i = (\ &(eth\_src = *, eth\_dst = *,\ eth\_type = *, vlan\_id = *, \\ &ip\_src = 130.192.225.11, ip\_dst = 8.8.8.113, ip\_proto = 0x06, \\ &port\_src = *,\ port\_dst = *), \{[< h\_a, H >, < vpn\_a, VPN >, \\ &< fw, FW >, < ws\_a, WS >], [< h\_a, H >, < vpn\_b, VPN >, \\ &< dpi, DPI >, < fw, FW >, < ws\_a, WS >]\}\ ) \end{aligned}$$

$$\begin{aligned} r_j = (\ &(eth\_src = *, eth\_dst = *,\ eth\_type = *, vlan\_id = *, \\ &ip\_src = 130.192.225.11, ip\_dst = 8.8.8.113, ip\_proto = 0x06, \\ &port\_src = *,\ port\_dst = *), \{[< h\_a, H >, < vpn\_a, VPN >, \\ &< fw, FW >, < ws\_a, WS >], [< h\_a, H >, < vpn\_b, VPN >, \\ &< dpi, DPI >, < fw, FW >, < ws_a, WS >], [< h\_a, H >, \\ &< vpn\_a, VPN >, < mn, MN >, < ws\_a, WS >]\}\ ) \end{aligned}$$

In this example, r_j contains an additional SFC with respect to r_i, but this kind of policy specification (even if it is ambiguous and non-optimized) may not be an anomaly because, for example, each of those chains contains a VPN functionality and the administrator does not want to be advertised in such cases.

In this model, we consider as pre-defined conflictual anomaly only the case when the two forwarding rules do not have any SFC in common, as defined in (2.18). All the other possible conflictual anomalies have to be specified by the administrators and are classified as custom anomalies (Table 2.2). An example of administrator-defined conflicting anomaly could be the case of a traffic flow that is managed by two forwarding rules that enforce "correlated" sets of SFCs (i.e., the two sets share some SFCs but they are not the same):

$$\begin{aligned}
&eth\_src_i = eth\_src_j \wedge eth\_dst_i = eth\_dst_j \wedge \\
&eth\_type_i = eth\_type_j \wedge vlan\_id_i = vlan\_id_j \wedge \\
&ip\_src_i = ip\_src_j \wedge ip\_dst_i = ip\_dst_j \wedge \\
&ip\_proto_i = ip\_proto_j \wedge port\_src_i = port\_src_j \wedge \\
&port\_dst_i = port\_dst_j \wedge \exists c_i^k \in \mathscr{C}_i, c_i^k \notin \mathscr{C}_j \wedge \\
&\exists c_i^y \in \mathscr{C}_i, c_i^y \in \mathscr{C}_j \wedge \exists c_j^l \in \mathscr{C}_j, c_j^l \notin \mathscr{C}_i \\
&\exists c_j^p \in \mathscr{C}_j,\ c_j^p \in \mathscr{C}_i \rightarrow CorrelatedChains(r_i, r_j)
\end{aligned} \tag{2.19}$$

2.4 Implementation and evaluation

In order to show the usefulness of the model proposed, we have implemented the proposed verification approach and tested it under a use-case topology, which represents a realistic campus network scenario. Our prototype implementation can work along with any SDN Controller that supports the network policy specification (e.g by means of policy-oriented languages or with other techniques). A possible future enhancement is to implement a verification module either integrated into an existing SDN Controller or located a layer above it.

Figure 2.3 shows our use-case, where each end-host (the use case includes about 300 hosts) generates several types of traffic flows (i.e., HTTP, POP3 and SMTP towards the internal servers and the Internet) that are processed by a number of network functions (in the magnitude of 10).

A Java-based prototype of the verification approach has been implemented, exploiting, as verification engine, the Drools tool. Drools is a Production Rule System [13], which is an expert system that uses a rule-based approach for reasoning. In particular, Drools processes an acquired knowledge into a knowledge base (i.e. Production Rules or simply Drools rules hereafter) to infer conclusions which result in actions. A Production Rule is a piece of knowledge that triggers an action when its condition is matched over the acquired knowledge. In our implementation, the acquired knowledge about the network scenario is implemented as Java objects, while an anomaly is represented by a Drools rule, where the triggered action implies the detection of the anomaly itself.

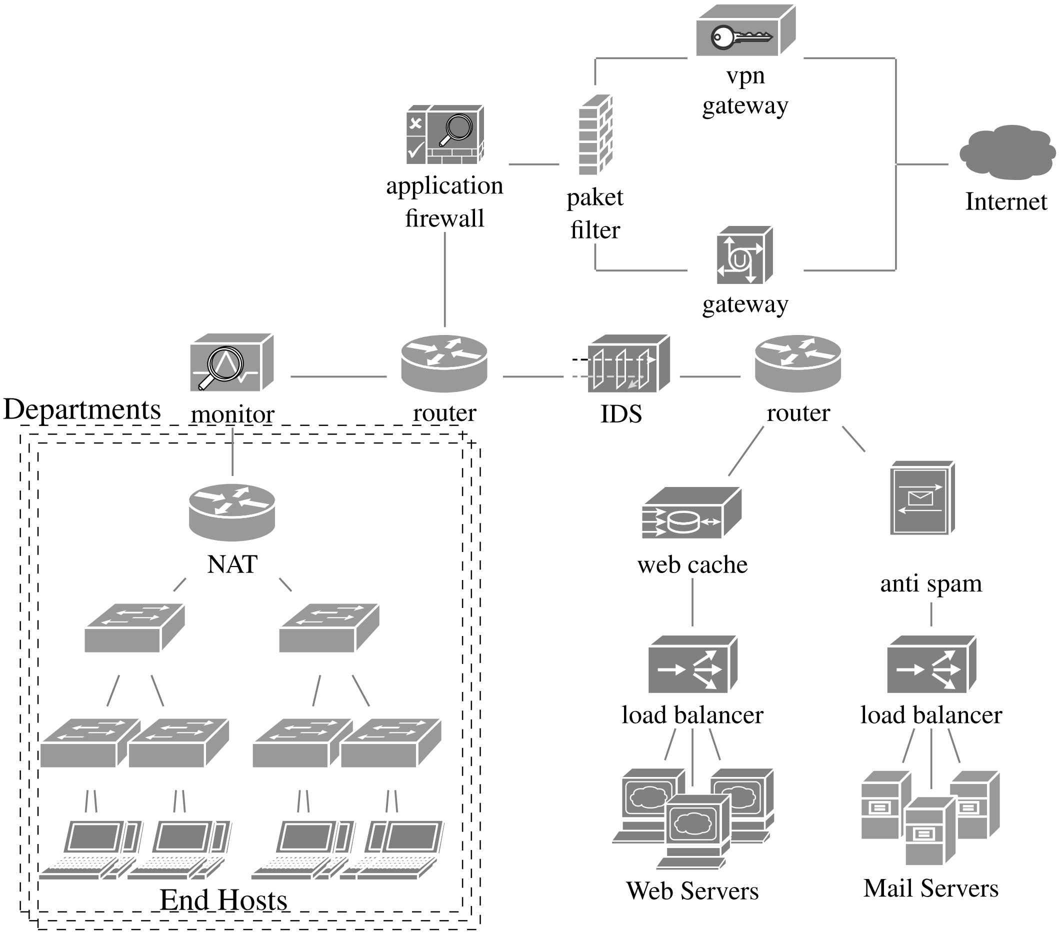

Fig. 2.3 Use case: a possible campus network topology.

The main purposes of this experimental evaluation were: *(i)* identifying the main factors that influence the time taken by this verification process and *(ii)* evaluating verification time as a function of the influential factors. In order to do this, we have performed our test scenarios on an Intel i7-4600U@2.10GHz workstation with 8 GB of RAM.

A set of forwarding rules has been automatically generated to represent a use-case as depicted in Figure 2.3. In particular, we have initialized each network field of the forwarding rule with a range of values when possible (e.g. port number or IP addresses), otherwise with exact values. In order to progressively increase the size of the forwarding rule set, the number of traffic flows has been increased, by considering wider and wider ranges of port number and traffic types and more subnets and hosts

Table 2.1 Pre-defined set of anomalies.

Class	Anomaly	Formula
Single-Field	bad *port_src* specification	see (2.12)
	bad *port_dst* specification	see (2.13)
	bad *vlan_id* specification	$vlan\_id < 1 \wedge vland\_id > 4094 \rightarrow BadVlanId(r)$
	bad *eth_type* specification	$eth\_type \neq \{0\mathbf{x}0800, 0\mathbf{x}0806, 0\mathbf{x}8100\} \rightarrow BadEthType(r)$
	bad *ip_proto* specification	$ip\_proto \neq \{0\mathbf{x}01, 0\mathbf{x}06, 0\mathbf{x}11\} \rightarrow BadIpProto(r)$
Pair-Field	equal source and destination IP addresses	see (2.15)
	equal source and destination Ethernet addresses	$eth\_src = eth\_dst \rightarrow BadEthernetAddress(r)$
Sub-Optimization	bad src port specification	see (2.9)
Conflicting	a single flow is forwarded to different chains	see (2.18)

Table 2.2 Custom set of anomalies.

Class	Anomaly	Formula
Node Traversal	web traffic does not traverse a web-cache	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.116 \wedge$ $ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge$ $port\_dst = 80 \wedge c^k \nsucc [<*, CACHE>],$ $\forall c^k \in \mathscr{C} \rightarrow WebNot2Cache(r)$
	mail traffic does not traverse an anti-spam	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.244 \wedge$ $ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge$ $port\_dst = 25 \wedge c^k \nsucc [<*, SPAM>],$ $\forall c^k \in \mathscr{C} \rightarrow MailNot2Spam(r)$
	web traffic traverses an anti-spam	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.116 \wedge$ $ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge$ $port\_dst = 80 \wedge c^k \succ [<*, SPAM>],$ $\forall c^k \in \mathscr{C} \rightarrow Web2Spam(r)$
	mail traffic traverses a web-cache	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.244 \wedge$ $ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge$ $port\_dst = 25 \wedge c^k \nsucc [<*, CACHE>],$ $\forall c^k \in \mathscr{C} \rightarrow Mail2Cache(r)$
	internet traffic does not pass through a L7 firewall	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 8.8.8.0/24 \wedge$ $ip\_proto = 0\mathbf{x}06 \wedge port\_dst = 80 \wedge c^k \nsucc [<*, L7\_FW>],$ $\forall c^k \in \mathscr{C} \rightarrow InternetNot2Firewall(r)$
Node Ordering	A firewall is not located after a NAT function but before it	see (2.16)
Chain Constraint	Internet traffic is not forwarded to correlated chains	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.116$ $\wedge ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge$ $port\_dst = 80 \wedge c^k \nsim c^l,$ $\forall c^k, c^l \in \mathscr{C} \rightarrow InternetNoCorrelatedChains(r)$
Conflicting	a single flow is forwarded to different chains	see (2.19)

per department (Figure 2.3). In this way, we have been able to test the scalability of the verification process.

For what concerns the set of anomalies checked at each test-run, we have considered both those anomalies the model supports by default (i.e., the pre-defined set that includes 16 anomalies) and custom anomalies, specific for the tested network scenario. In detail, our forwarding rule set has been generated in such a way to trigger at least one anomaly for each class presented in Section 2.3. The whole

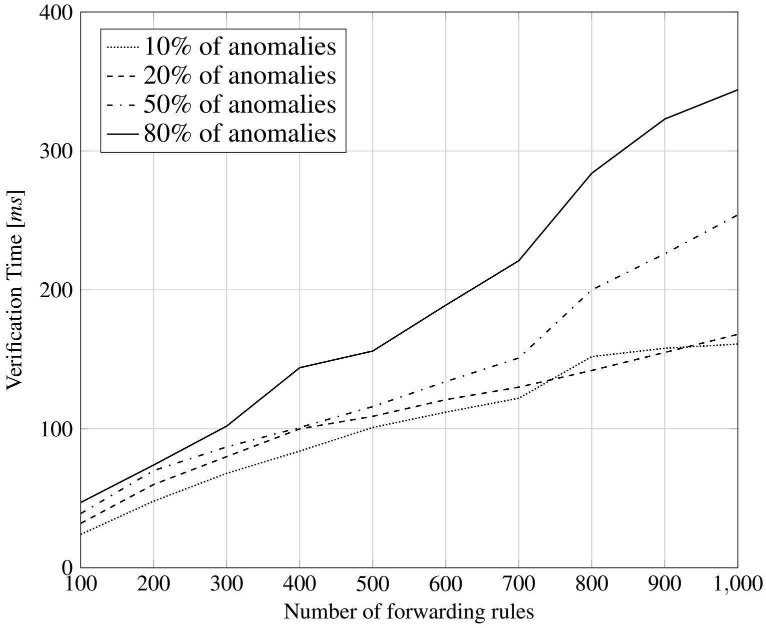

Fig. 2.4 Verification time evaluated with a growing number of forwarding rules.

set of anomalies that has been checked in our network scenarios is summarized in Tables 2.1 and 2.2, where, for each anomaly, we present a possible formula that detects that anomaly for a specific flow.

Moreover, in the automatic generation process, we have set a threshold on the percentage of forwarding rules (with respect to the total rule set) that trigger an anomaly. Figure 2.4 shows that, for each rule set-size, we have evaluated the elapsed time in case of 10%, 20%, 50% and 80% of "anomalous" rules.

The obtained results indicate that the elapsed time to complete the verification scenario grows linearly with the number of forwarding rules. This is highlighted in the four test scenarios. The measured times have been averaged on 100 test-runs and have a magnitude of 340ms in the worst case (the solid line in Figure 2.4).

In order to check if the verification time depends also on the percentage of forwarding rules that trigger an anomaly, we have performed another type of test (Figure 2.5). Keeping constant the number of forwarding rules, we have evaluated the verification time by growing the percentage of anomalies from 0% to 100%. Under these conditions, we have verified four test scenarios shown in Figure 2.5, where the number of forwarding rules has been set at 100, 300, 500, 700 and 1000 rules.

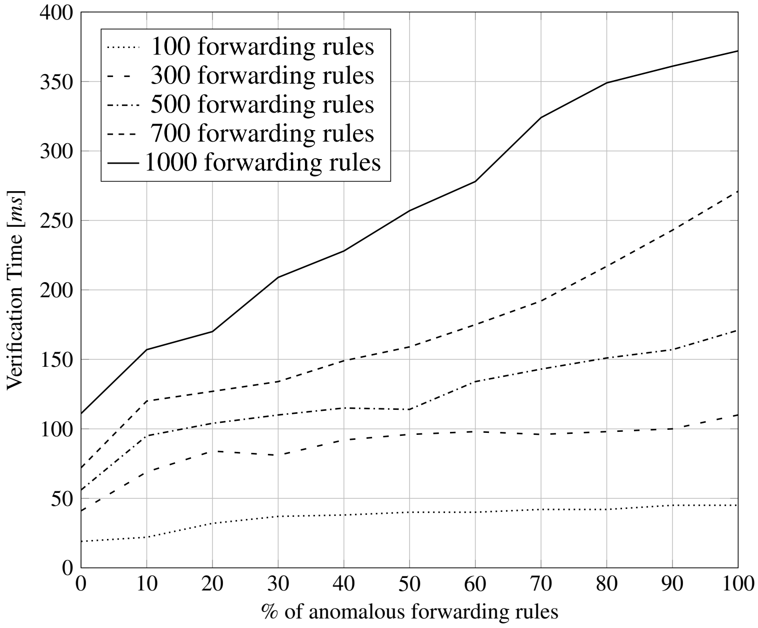

Fig. 2.5 Verification time evaluated with a growing percentage of forwarding rules that satisfy an anomaly.

Also in this second case, we have evaluated the verification time 100 times for each size of the rules set. As we can note from the achieved results (Figure 2.5), the percentage of forwarding rules that bring to an anomaly has a major influence on verification time when the rule set size grows. This can be confirmed by comparing the trend in case of 100 forwarding rules, where the elapsed time is quite constant, and in case of 1000 rules, which arises rapidly with the increment of the anomaly percentage.

The achieved results are also confirmed in an additional test case (Figure 2.6), where we have evaluated the verification time in case of a growing number of "anomalous" rules in different sized rule-sets (i.e., 100, 300, 500, 700 and 1000 forwarding rules). Also in this test-scenario, it is evident that the performance of our verification module is influenced by both the number of forwarding rules and the percentage of these that trigger an anomaly.

Moreover, we can also note that the verification time is ranged between 350ms, in the worst case with 1000 forwarding rules and 80% of "anomalous" rules (Figure 2.4), and 400ms, when each of the 1000 forwarding rules trigger an anomaly (i.e., the solid lines in Figure 2.5 and Figures 2.6).

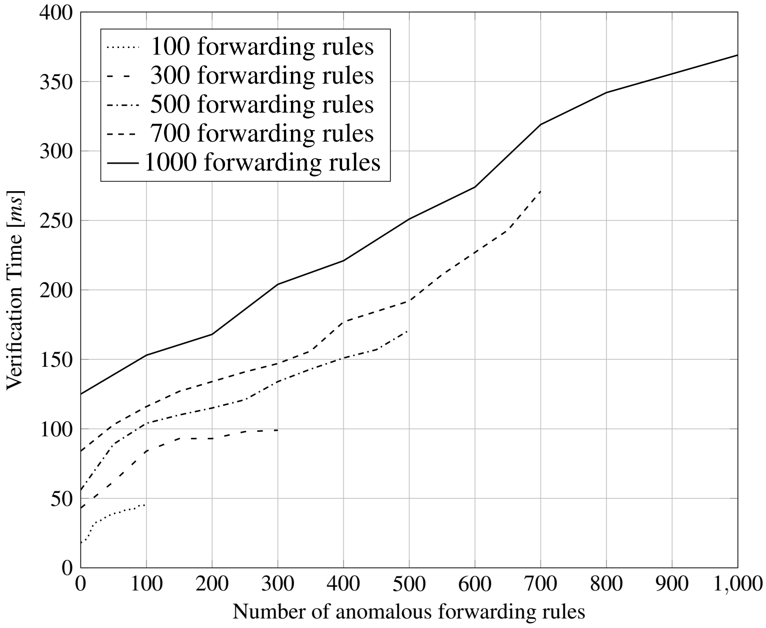

Fig. 2.6 Verification time evaluated with a growing number of anomalies.

The achieved results show that our verification approach takes a time in the order of hundreds of milliseconds in the case of a real-sized network with a growing number of traffic flows, so that it is reasonable to use our approach in a real network scenario.

2.5 Related Work

In this section, we provide an overview of the main works presented in the literature that have completed our background to improve the state of the art about the detection of anomalies in service function chains defined as forwarding policies.

We started with the most relevant proposals related to SDN verification. As we already mentioned, differently from our work on early-verification of forwarding policies, this part of the literature is mainly related to OpenFlow networks, where it checks the presence of conflicts in FlowTables and the violation of network invariants (e.g black holes and forwarding loops).

Current literature on SDN verification generally adopts two approaches: either off-line or real-time verification [14]. The first-generation of OpenFlow-oriented verification tools worked off-line, which means they take a snapshot of the global network behaviour (i.e., by collecting the forwarding entries installed into the network switches) and check whether some basic invariants hold. A first example of off-line tool is NICE [15], which checks the presence of network invariants by combining model checking and symbolic execution approaches. Similarly, Anteater [16] verifies such invariants by expressing switch configurations as boolean satisfiability (SAT) problem instances. Another example is also NetPlumber [17], which relies on the HSA (Header Space Analysis) [18] that allows for static checking of the OpenFlow rules of a whole network to detect forwarding loops and leakage problems, by exploiting Network Transfer Functions.

The second approach (i.e. real-time) was to locate the verification tool as a layer between the SDN Controller and network switches in order to check at run-time the violation of some invariants. This is the case of VeriFlow [7], which dynamically checks the absence of forwarding loops and black holes is satisfied at each OpenFlow rule insertion. Another example of real-time verification tool is FlowChecker [8], which is also able to simulate different components during the test execution. Since the network behaviour is dictated by the user's configuration and in case of misconfiguration the whole network lead into an unsafe state, FlowChecker helps to detect security attacks by checking periodically the authenticity of the configuration parameters selected by the user.

Locating a layer between the SDN Controller and the network switches means that the amount of network traffic to check could be very big and could effect in a negative way the network performance. However, for what concerns the purposes of this thesis, the main weakness of such OpenFlow-oriented tools is that they can check only some components in the network (i.e., network switches). This implies that SDN-oriented tools verify if some invariants hold by checking only the flow entries in SDN switches (regardless of whether such tools work at run-time or not), causing a late detection of errors and faults in the service graph requests.

Moreover, recent works proposed policy-based strategies for instructing SDN/OpenFlow Controllers on how traffic flows should be forwarded into the network. The Controller will install the consequent flow entries in network switches to enforce the forwarding policies.

Under this umbrella, there are Merlin [19], a unified framework for enabling administrators to define forwarding policies with bandwidth constraints, and Fat-

Tire [6], a NetCore-based language to describe SFCs and specify fault-tolerance requirements.

For what concerns policy specification, our approach has some common points with the Merlin and FatTire languages. Similarly to these two works, we have classified traffic flows by means of network fields and we have specified SFCs, listing the network functions involved into the chains. However, the existing proposals such as Merlin and FatTire do not have an underlying formal model for detecting anomalies. They include a form of verification, but it is not fully documented and, at the best of our knowledge, it is not as intensive as we proposed. While we focus on checking several classes of anomalies, also enabling administrators to define their own anomalies, it seems that Merlin includes only the verification that any sub-policy modification introduced by tenants does not violate the global policy set down by the administrator. In particular, this verification consists in checking if the policy modification includes the chains enforced in the original policy.

FatTire instead generates configurations that are correct-by-constructions, because it verifies the forwarding policies by normalizing them into a set of non-overlapping atomic policies at compile phase. The non-overlapping policies will then be exploited to create the final OpenFlow configurations.

Finally, we have proposed a unified forwarding policy abstraction to represent the fundamental features of policy-oriented languages (i.e., classifying traffic flows and steering packets), in order to avoid a verification model specific for each language. This verification model can be integrated into any policy-oriented SDN Controller, with the addition of a language-specific module to generate the unified verification model from the high-level language. However we are mainly interested in presenting the details of the verification model and its features, while the design of a translation module is left as future work.

Chapter 3

Checking Reachability in Service Graphs

From a provider point of view, the integration of a checking mechanism during the deployment of a service graph is a step forward in the quality of the services offered to the final users: checking the correctness of the service requests can reduce the risk of unexpected network behaviour at run-time and can limit service downtime.

In order to integrate network verification in the process to deploy a service graph, we recall providers need to face some challenges, like: *(i)* perform verification before deploying the service graphs and, in case of failure, stop the graph instantiation; *(ii)* check graph correctness in a reasonable amount of time from a VNF Orchestrator point of view, with fair processing resources (e.g., CPU, memory, etc...). These requirements are needed because we are in the context of flexible services, where the reconfiguration of network can be frequently triggered, both in case of user request and in case of management events. Thus a provider needs to detect errors and faults in the service graph definition early, in order to speed up the fixing phase, and quickly, in order to avoid to impact the quality of experience of the service offered to the final users.

Even though a plethora of solutions has been presented in the literature (see Section 2.5), such tools check only the configuration of SDN switches/routers, overlooking other types of VNFs able to modify the forwarding behaviour of the

[1] This work is published in [20] and partially described in the PhD thesis of Matteo Virgilio [21], who collaborated in the development of the approach.

network at run-time. In particular, we refer to the presence of *stateful network functions* (or also *middleboxes*) in the network, which forward packets based on both the packet content and their internal algorithm and state, like a Network Address Translator or a Deep Packet Inspection can do.

In order to improve the literature, we propose a way to model stateful middleboxes[2] involved in the network and their functional configurations (e.g., filtering policy for a firewall) and a checking mechanism to verify the satisfaction of reachability-based invariants in service graph requests.

3.1 Problem statement and contributions

Ultra broadband diffusion, progresses in Information Technologies (IT), tumbling hardware costs and a wider and wider availability of open source software are shaping the evolution of Telecommunications and ICT infrastructures. Thanks to the already introduced concept of “Softwarization” of the network, it is argued that future Telecommunications infrastructures are likely to become highly dynamic, flexible and programmable production environments of ICT services.

Due to these reasons, we aim at investigating novel approaches of network verification (i.e., the definition of methods and techniques to validate a particular network configuration before deploying it) in line with the providers requirements. We recall in fact that a verification process can be seen as an essential task in environments where reconfiguration of services is expected to be triggered very frequently, both in response to user requests and also in case of management events. Misconfiguration of dynamic network middleboxes, violation of specified network policies, or artificial insertion of malicious network functions are just examples of cases that a complete solution must properly handle in order to preserve network integrity and reliability. The adopted verification process thus must prefer scalable and fast checking techniques instead of complex modeling approaches in order to take reasonable verification time.

For this reason, one of our contributions goes in the direction of verifying complex service graph through an intense modeling activity, targeted at the specific middleboxes and the network as a whole. We also aim at integrating our verification

[2] In this thesis, we use the terms VNF, network function and middlebox interchangeably.

approach of into one of the existing SDN/NFV-based environments, i.e. the one carried out within the EU FP7 UNIFY[3] consortium, which sets out to integrate modern cloud computing and networking technologies by considering the entire network as a unified service production environment, spanning the vast networking assets and data centers of telecom providers.

In order to reach a high level of agility for service innovation, UNIFY has one focus on providing dynamic service programming and orchestration, deploying logical service components, i.e. VNFs, across multiple network nodes. In particular, UNIFY architecture follows SDN principles with a logically centralized control and orchestration plane. Additionally, compute, storage and network abstractions are combined into a joint programmatic interface referred to as Network Function Forwarding Graph (NF-FG). An NF-FG is a low-level definition of a Service Graph, which defines a selected mapping of VNFs and their forwarding overlay definition into the virtualized resources presented by the underlying layer. From a verification point of view, we did no matter of the virtual resource assignment. Thus, in this thesis, we consider service graphs equivalent of their orchestration/infrastructure-level description, i.e. NF-FGs.

Current OSS/BSS do not seem to cope with the requirements posed by this evolution: in fact, the operations of future Telecommunications infrastructures will involve the management and control of a myriad of software processes, rather than closed physical nodes. Thus, another important goal of UNIFY is the design and development of integrated operations and development capabilities under the name of Service Provider-DevOps (SP-DevOps). In fact, DevOps paradigm, formerly developed for Data Centers (DCs), is getting momentum as a source of inspiration regarding how to simplify and automate management processes for future Telecommunications infrastructures.

As we have already mentioned, we are motivated by the observation that most existing tools are OpenFlow-oriented, i.e. they mostly consider networks with a controller which installs `<match, action>` rules on the switches. Alternatively (and more generically but with the same fundamental limitations), they consider networks with devices that only perform forwarding decisions according to the packet header, i.e. without taking into account any additional traffic history information. We recall that works as [7, 18, 22, 23] (Section 2.5) fall in this category and represent a

[3] www.fp7-unify.eu

valuable efforts in this research area. Our contribution is intended to move a step forward and overcome the above mentioned limitations by extending these works. In this sense, one important reference is [24], which tackles exactly the same problem and provides a scalable solution based on an off-the-shelf SMT solver.

We experiment with this approach and further develop it to meet our specific requirements, also enriching the available VNF models catalog in order to satisfy the demands for more and more complex service graphs and to validate the approach with different kinds of VNFs. We specifically consider the UNIFY use cases, but it is worth noticing how our work is much general and easily applicable to other scenarios since it involves very common network functions.

The rest of the chapter is organized as follows. First, we introduce and clarify how and to which extent the DevOps approach can be applied in a network operator infrastructure (Section 3.2). After defining the processes needed to implement this vision, we move on our current approach to formally verify complex and rapid deployments of network function chains including a variety of middleboxes, deployed to augment the set of in-network services the operator is able to offer to its final customers (Section 3.3). In order to show our approach is feasible, we provide some preliminary performance evaluation results based on the extension of the above mentioned tool (Section 3.4).

3.2 The SP-DevOps concept

In order to cope with the high service velocity and increased dynamicity enabled by current SDN/NFV-based environments (e.g. UNIFY), we consider a novel management and operation paradigm for Service Providers, called Service Provider DevOps (SP-DevOps), and how it has been applied by UNIFY.

SP-DevOps is based on the same major underlying principles as identified for DevOps [25]: *(i)* Monitor and validate operational quality; *(ii)* Develop and test against production-like systems; *(iii)* Deploy with repeatable, reliable processes; and *(iv)* Amplify feedback loops.

While we acknowledge that DevOps has also a crucial cultural dimension (reflected barely by the feedback loop principle), UNIFY focused on technical aspects associated to these principles, which reflect on processes and associated capabili-

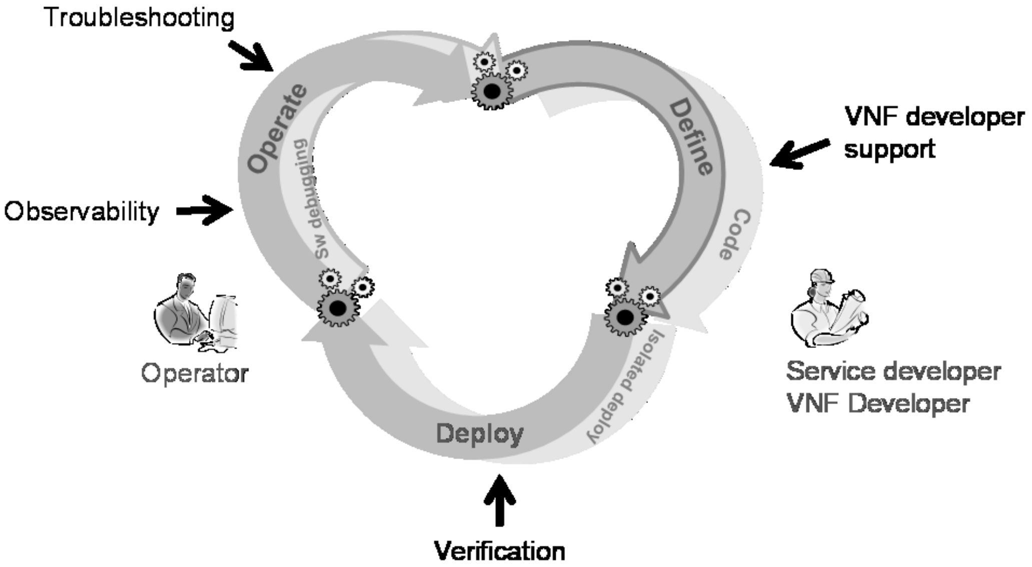

Fig. 3.1 SP-DevOps cycle for UNIFY service creation.

ties for integrated monitoring, verification, and testing software and programmable infrastructure.

Even if significant parts of the telecommunication networks are foreseen to be virtualized in the future, [26] identified important characteristics of telecommunication networks that differ from traditional data centers, i.e.:

(i) higher spatial distribution, as telecom resources are spread over wide areas due to coverage requirements;

(ii) lower levels of redundancy in access and aggregation networks compared to the massive data centers of typical cloud computing companies;

(iii) stronger requirements on high availability and latency in according to standards and customer expectations.

These characteristics pose new challenges for applying DevOps principles in telecommunications environments [27]. SP-DevOps addresses them with a set of technical processes supporting developer and operator roles in a virtualized telecom network. Figure 3.1 illustrates the relation between the four SP-DevOps processes and the developer/operator roles by means of a service creation lifecycle. The four SP-DevOps processes follow the DevOps principles to meet specific challenges regarding Observability and Troubleshooting (Principle: Monitor and validate operation quality); Verification (Principle: Deploy with repeatable, reliable processes); and Development (Principle: Develop and test against production-like systems). Three main roles are also involved in the processes: two Developer roles, where one is associated to a classical operator role assembling the service graph for a particular category of services (the Service Developer), and a second associated to the classical equipment vendor role in actually programming a VNF (the VNF Developer). The role of the Operator is to ensure that a set of performance indicators associated to a service are met when the service is deployed on virtual infrastructure within the domain of a telecom provider.

SP-DevOps might not be a new form of DevOps as such, but it must include solutions that are uniquely tailored for the characteristics of its environment. Consequently, UNIFY proposed the SP-DevOps Toolkit as an instantiation of the SP-DevOps concept [28]. The SP-DevOps Toolkit[4] consists of a set of DevOps solutions that are developed targeting specific research challenges identified in the UNIFY production environment [27, 26]. Besides scalable and programmable infrastructure monitoring functions, the toolkit will also provide modules for deploy-time functional verification of various abstraction levels of service definition, supporting the three SP-DevOps roles.

As in any development process, identification of problems early in the service or product lifecycle can significantly reduce times and costs spent on complicated debugging and troubleshooting processes.

In this thesis, we focus on verification with respect to the service definitions and configurations initiated by the Service Developer. Automated verification functions operating during deploy-time on each layer of the orchestration and control architecture, facilitate verification as part of each step in the deployment process, allowing identification of problems early in the service lifecycle. We present in next sections our verification approach, which has been released both as open-source tool and as part of the SP-DevOps Toolkit developed within UNIFY.

[4] http://www.fp7-unify.eu/index.php/results.html#OpenSource

3.3 The approach

The SP-DevOps paradigm represents a significant opportunity for service providers to implement more complex services in their networks and increase the agility by which a new function (or a chain of) can be automatically configured and deployed in their infrastructure. However, while the process of inserting and/or modifying functions throughout the network can be automated with technologies similar to the ones used for the Cloud Computing scenario [29], great importance has also to be placed on the design and implementation of automatic tools that can verify a network configuration on the fly, *before* it is deployed. To achieve this goal, we have to design a verification approach that is not limited to OpenFlow networks and that overcomes the limitations of existing tools (i.e. complex modelling approaches, considerable verification time and resources).

An example of configuration we aim to check is tha case of an operator that may want to ensure that a given traffic flow is permitted (or not permitted, due to a policy constraint) from one node to another. Concerning this last aspect, our verification process is currently based on a verification approach recently proposed in [24]. In order to achieve high performance, this verification approach exploits Z3 [30], a state of the art SMT solver, and translates network scenarios with multiple middleboxes into sets of First Order Logic (FOL) formulas that are then analyzed by Z3. This choice is motivated by the overall verification tool performance and scalability, which would be hard to achieve with standard model checking based techniques. In fact, the latter requires time and memory that usually increase exponentially with the system complexity, while the SAT-based approach proposed in [24] seems to be less prone to this problem.

The FOL formulas given to Z3 represent the network operating principles along with the functional behavior of all the VNFs involved in the scenario being considered. While [24] presents the general ideas of the proposed approach, not all the details are fully developed, and not all the different situations that may arise when considering different kinds of VNFs are considered.

Here, we present our preliminary work towards integrating the approach presented in [24] into a SP-DevOps context like the one of UNIFY. A considerable part of this work has been about improving the modeling approach for developing models of new VNFs that were not explicitly considered in [24] in order to be able to check

more complex and realistic network scenarios, and making some first experiments with such new models.
In our design, the formal verification task is split into multiple sub-tasks, so that the whole process is simpler and faster. More precisely, at NF-FG deploy time, or when the graphs undergo modifications in response to higher level events (e.g., administration events or user requests), the VNF chains composing the graph are computed and then, for each of them, a formal model is generated, including the model of all the involved VNFs.

The verification engine then processes the whole VNF chain model to check the satisfiability of a given property and in case of failure it stops the deployment phase. The verification process is applied for each chain that is included in the NF-FG, which will be deployed only in case the verification process ends with a successful result.

For what concerns the deployment of a service request, some of the required steps are: *(i)* generation of low-level configurations (e.g. flow entries of OpenFlow FlowTables) from the NF-FG specification and their deployment into the network switches and routers to instantiate the desired service chains; *(ii)* selection of the most suitable VNF instances to implement the requested service graph; *(iii)* VNF deployment into the network and resource assignment; *(iv)* etc....

In this thesis, we do not consider to translate the VNF models adopted in the verification approach into low-level configurations (e.g. OpenFlow rules), even though it may be possible. This is because, in this case, the services offered to the users may be limited to the capability of the Southbound protocol (e.g. OpenFlow) to implement at run-time the behaviour of a new network function. However, we are mainly interested in the verification aspects of a service graph and thus we do not present all the steps required to complete its deployment and how they work.

In this chapter, we focus on reachability problems in service graphs, leaving the verification of other network properties as possible future work. Furthermore, since we are using abstract models of the real middleboxes, we assume that these models are correctly defined. This means that we verify abstract models of the real middleboxes, considering them as faithful representations of the real VNFs. Verification of possible mismatch between a VNF model and its implementation is out of scope for the current prototype.

3.3.1 VNFs models

The approach for modeling network function chains proposed in [24] has been experimented by the authors with some middlebox types, such as stateless and stateful firewalls.

When modelling scenarios that include VNFs that may alter packets (e.g., a NAT), it is necessary to also consider the possibility for a target VNF to receive a packet different from the one originally transmitted. This kind of situation regards a significant set of middleboxes that is currently deployed in SP networks and that is envisioned to be included in the NF-FG within the UNIFY project, e.g. NAT, VPN gateway and so on.

One of our contributions to the state of the art was to revise the network constraints developed by the authors of [24], by introducing the possibility of verifying reachability properties between two network nodes and intermediate VNFs that do modify forwarded packet headers. We have also checked that verification works as expected with these revisited constraints, by experimenting with the new middlebox models that we developed. Thanks to these improvements, providers will be able to check an enriched set of realistic network scenarios, improving the quality of service offered to the final users.

The first VNF we consider is a simple web cache (reported in Figure 3.2). The functional model consists of two interfaces connected respectively to the private network, i.e., the one which contains the clients issuing HTTP requests, and the external network.

Formula 3.2a states that a packet sent from the cache to a node belonging to the external network, implies a previous packet, containing a HTTP request and received from an internal node, which cannot be served by the cache (otherwise the request would have not been forwarded towards the external network). Formula 3.2b states that a packet sent from the cache to the internal network contains a `HTTP RESPONSE` for an URL which was in cache when the request has been received. We also state that the packet received from the internal network is a `HTTP REQUEST` and the target URL is the same as the response. The final formula expresses a constraint that the *isInCache()* function must respect. In particular, we state that a given URL (u_0) is in cache at time t_0 if (and only if) a request packet was received at time t_1 (where $t_1 < t_0$)

$$
\begin{aligned}
&(send(cache,n_0,p_0,t_0) \wedge \neg isInternal(n_0)) \implies \neg isInCache(p_0.url,t_0)\wedge \\
&\quad p_0.proto = HTTP\_REQ \wedge \exists(t_1,n_1) \,|\, (t_1 < t_0 \wedge isInternalNode(n_1)\wedge \\
&\quad recv(n_1,cache,p_0,t_1)), \forall n_0,p_0,t_0
\end{aligned} \tag{3.2a}
$$

$$
\begin{aligned}
&(send(cache,n_0,p_0,t_0) \wedge isInternal(n_0)) \implies isInCache(p_0.url,t_0)\wedge \\
&\quad p_0.proto = HTTP\_RESP \wedge p_0.ip\_src = p_1.ip\_dest\wedge \\
&\quad p_0.ip\_dest = p_1.ip\_src \wedge \exists(p_1,t_1) \,|\, (t_1 < t_0\wedge \\
&\quad p_1.protocol = HTTP\_REQ \wedge p_1.url = p_0.url\wedge \\
&\quad recv(n_0,cache,p_1,t_1)), \forall n_0,p_0,t_0
\end{aligned} \tag{3.2b}
$$

$$
\begin{aligned}
&isInCache(u_0,t_0) \implies \exists(t_1,t_2,p_1,p_2,n_1,n_1) \,|\, (t_1 < t_2 \wedge t_1 < t_0\wedge \\
&\quad t_2 < t_0 \wedge recv(n_1,cache,p_1,t_1) \wedge recv(n_2,cache,p_2,t_2)\wedge \\
&\quad p_1.proto = HTTP\_REQ \wedge p_1.url = u_0 \wedge p_2.proto = HTTP\_RESP\wedge \\
&\quad p_2.url = u_0 \wedge isInternal(n_2))\forall u_0,t_0
\end{aligned} \tag{3.2c}
$$

Fig. 3.2 Web cache model.

for that URL and a subsequent packet was received at time t_2 (where $t_2 < t_0 \wedge t_2 > t_1$) carrying the corresponding `HTTP RESPONSE`.

The second middlebox we modeled is the NAT function. The corresponding model is reported in Figure 3.3. In order to model the NAT behaviour, a distinction between the private and external network is needed. This separation is modeled by using a boolean function (*isPrivateAddress()*) that returns true if a given IP address belongs to the set of internal node addresses.

Analyzing the reported formulas, we start by considering an internal node which initiates a communication with an external node (Formula 3.3a). In this case, the NAT sends a packet (p_0) to an external IP address, if and only if it has previously received a packet (p_1) from an internal node. The received and sent packets must be equal for all fields, except for the *ip_src*, which must be equal to the NAT public IP address.

On the other hand, the traffic in the opposite direction (from the external network to the private) is modeled by the Formula 3.3b. In this case, we state that if the NAT is sending a packet to an internal address, this packet (p_0) must have an external IP address as its source. Moreover, p_0 must be preceded by another packet (p_1 in the formula), which is, in turn, received by the NAT and it is equal to p_0 for all the

$$\begin{aligned}
&(send(nat,n_0,p_0,t_0) \land \neg isPrivateAddress(p_0.ip\_dest)) \implies \\
&\quad p_0.ip\_src = ip\_nat \land \exists(n_1,p_1,t_1) \mid (t_1 < t_0 \land recv(n_1,nat,p_1,t_1)\land \\
&\quad isPrivateAddress(p_1.ip\_src) \land p_1.origin = p_0.origin\land \\
&\quad p_1.ip\_dest = p_0.ip\_dest \land p_1.seq\_no = p_0.seq\_no\land \\
&\quad p_1.proto = p_0.proto \land p_1.email\_from = p_0.email\_from\land \\
&\quad p_1.url = p_0.url)\forall n_0,p_0,t_0
\end{aligned} \tag{3.3a}$$

$$\begin{aligned}
&(send(nat,n_0,p_0,t_0) \land isPrivateAddress(p_0.ip_dest)) \implies \\
&\quad \neg isPrivateAddress(p_0.ip\_src) \land \exists(n_1,p_1,t_1) \mid (t_1 < t_0\land \\
&\quad recv(n_1,nat,p_1,t_1) \ \land \neg isPrivateAddress(p_1.ip\_src)\land \\
&\quad p_1.ip\_dest = ip\_nat \ \land p_1.ip\_src = p_0.ip\_src\land \\
&\quad p_1.origin = p_0.origin \land p_1.seq\_no = p_0.seq\_no\land \\
&\quad p_1.proto = p_0.proto \land p_1.email\_from = p_0.email\_from\land \\
&\quad p_1.url = p_0.url) \ \land \exists(n_2,p_2,t_2) \mid (t_2 < t_1 \land recv(n_2,nat,p_2,t_2)\land \\
&\quad isPrivateAddress(p_2.ip\_src) \ \land p_2.ip\_dest = p_1.ip\_src\land \\
&\quad p_2.ip\_dest = p_0.ip\_src \land p_2.ip\_src = p_0.ip\_dest),\forall n_0,p_0,t_0
\end{aligned} \tag{3.3b}$$

Fig. 3.3 NAT model.

other fields. It is worth noting that, generally, a communication between internal and external nodes cannot be started by the external node in presence of a NAT. As a consequence, this condition is expressed in the Formula 3.3b by imposing that p_1 must be preceded by another packet p_2, sent to the NAT from an internal node.

3.4 Implementation and evaluation

In this section, we present further details on how we have implemented and tested the verification approach presented previously. Encouraged by our preliminary results (Section 3.4.1), we further develop our prototype in a service-oriented architecture that enables providers to perform a full verification process (Section 3.4.2). The last part of this section will present further results of how our tool works integrated in a provider environment as the UNIFY architecture (Section 3.4.3).

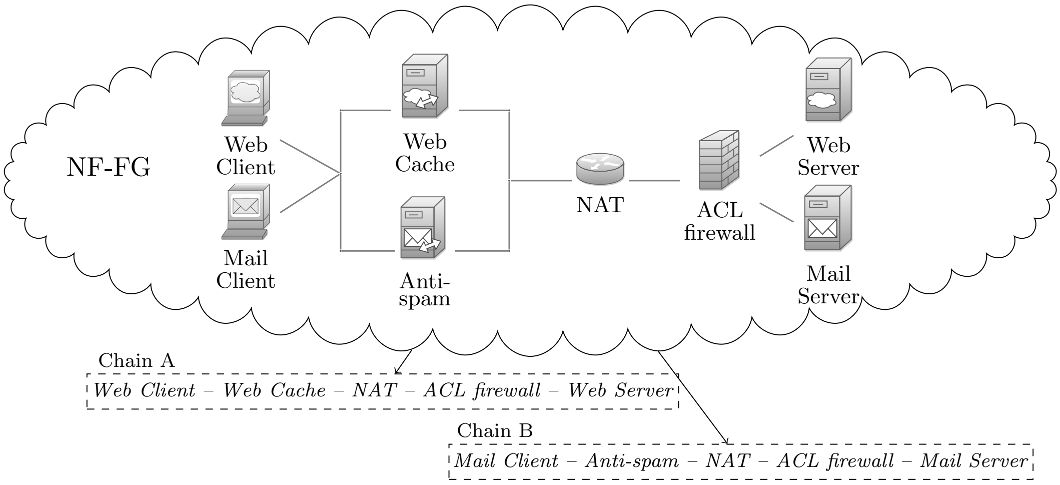
NF-FG
Web Client
Mail Client
Web Cache
Anti-spam
NAT
ACL firewall
Web Server
Mail Server
Chain A
Web Client – Web Cache – NAT – ACL firewall – Web Server
Chain B
Mail Client – Anti-spam – NAT – ACL firewall – Mail Server

Fig. 3.4 An example of Network Function-Forwarding Graph.

3.4.1 Preliminary results

In order to check if the new developed models and the overall approach fulfil the aforementioned verification constraints (i.e. reasonable times and resource usage), we consider the NF-FG[5] shown in Figure 3.4 as a use case.

In our reference graph, four end-hosts (two clients and two servers) can generate either HTTP or POP3 and also SMTP traffic, which is processed by different middle-boxes when traversing the graph. Moreover, some of those network functions may require a different configuration. Specifically, the NAT must be configured in order to know which hosts belong to the private network (as the web cache) and which IP address must be used as masquerading address; the firewall must be provided with a set of ACL entries that specify which couples of nodes are authorized to exchange traffic. Additionally, the forwarding is configured such that the web traffic is forwarded to the web cache, while the email traffic (both POP3 and SMTP) is routed to an anti-spam function. A first step towards the NF-FG verification is the VNF chains extraction. In our use case, two chains are extracted from the NF-FG (Figure 3.4): the *Chain A* processes the web traffic, while the *Chain B* is traversed by POP3 and SMTP packets.

We perform multiple tests on the two chains to cover different cases and configuration options: *(i)* anti-spam and firewall configurations and *(ii)* traffic directions (from client to server and vice-versa). Concerning the Chain A, only the ACL firewall

[5] We do not provide the firewall VNF model as it was presented as use case in [24].

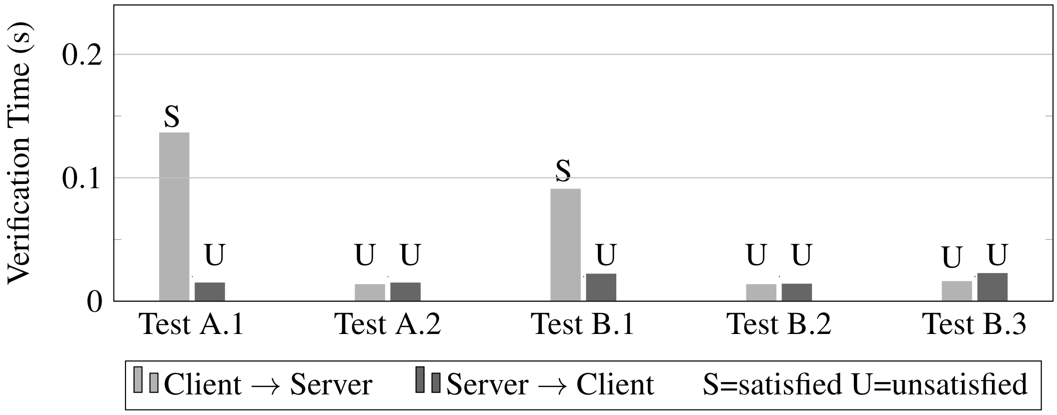

Fig. 3.5 Formal verification of a service graph with stateful VNFs.
Test {A, B}.1: firewall and anti-spam configured to accept packets; **Test {A, B}.2**: firewall configured to drop server/client packets; **Test B.3**: anti-spam configured to drop server/client packets.

can be configured, hence we setup two tests: one with the firewall configured to allow all the traffic (test A.1) and the other one with the firewall configured to drop all packets exchanged between the web client and server (test A.2).

Instead the Chain B is tested in three scenarios, obtained by changing the firewall and anti-spam configurations as follows: (*i*) test B.1, similarly to test A.1, is performed without any function configured to drop the received traffic; (*ii*) in test B.2, the firewall drops the traffic between the mail client and server (Figure 3.5); (*iii*) test B.3 is such that the anti-spam is configured to drop all the emails sent by the mail client, while the traffic originated by the server is allowed (Figure 3.5).

Our evaluation is executed on a workstation with 32GB of RAM and an Intel i7-3770 CPU running an Ubuntu 14.04.01 with kernel 3.13.0-24-generic. The results are shown in Figure 3.5, where the verification time is reported for each presented scenario.

In test A.1 the reachability problem from the client to the server (the light grey colored bar in Figure 3.5) is satisfied as expected. It is worth noting that the unsatisfiability of the problem in the opposite direction (the dark grey colored bar in Figure 3.5) is due to the fact that client and server can exchange traffic only if the connection is initiated by the client. In test A.2, in both cases the reachability problems are not satisfied because of the firewall VNF configuration. In test B.1, the verification problem is satisfiable in case of traffic sent by the mail client, while the reachability property is not verified for the traffic sent by the mail server for the above-mentioned reasons.

As it can be seen from the achieved results, performance is promising also in the worst case scenario, since we are able to solve the reachability problem in less than 200ms, while the verification time is less than 50ms in most cases. This is reasonably in line with the requirements dictated by a SDN/NFV environment like UNIFY, especially in terms of time required by the verification process to authorize a newly asked network reconfiguration.

3.4.2 VeriGraph

VeriGraph is the Java implementation of the already presented verification approach, released as open source tool[6] under the AGPLv3 license. We recall that the verification engine exploited in this work is the SMT solver Z3, developed by Microsoft. Moreover, VeriGraph relies on a graph-oriented database for managing service graph storage, that is Neo4J[7].

In details, the whole verification process is performed by the synergistic collaboration of different modules that interact with one another by means of well-defined interfaces [31]. Since the whole verification process is split into multiple sub processes, we designed a modular architecture to dispatch and execute the work accordingly.

In order to modularize the design of the overall tool and ease the integration with the UNIFY architecture, we designed the components shown in the verification tool in Figure 3.6 to perform the above-mentioned tasks.

Just to give an overview of the UNIFY architecture, it has a three-layered design to enable different levels of network abstraction: the service layer is aware of the service logic and it receives the service graph as service description; the second level hosts the orchestration logic to map the service graph requests to the available resources in the underlying level (i.e., the NF-FG) and to optimize; finally the infrastructure layer contains physical and virtual resources (i.e., compute, storage and networking resources). In particular, we have integrated our verification tool into the reference prototype implementation of the UNIFY architecture, which is ESCAPEv2 [32]. ESCAPE incorporates all the three layers of network abstraction

[6] https://github.com/netgroup-polito/verigraph
[7] https://neo4j.com/

envisaged by UNIFY (Figure 3.6) and provides a common platform to control and configure each step of the SFC lifecycle.

Essentially, our verification tool consists of two logical modules:

(i) VeriGraph is the core component and it is externally exposed by means of a RESTful API. Its fundamental role is that of receiving the service graph under deployment from the VNF Orchestration Layer and decompose it into different function chains in order to perform reachability-based verification on each involved chain. In this context, the term “function chain” is used to indicate a non-cyclic sequence of middleboxes that starts from one source node and ends into a different destination node. As a consequence, starting from a given service graph, it is generally possible to extract multiple chains with the same source and destination nodes. VeriGraph also receives the invariants to check against the requested service graph;

(ii) Neo4JManager[8] is used internally by VeriGraph to store the under deployment graph onto the Neo4J database and extract all the required chains from the graph based on the reachability invariants to check. This module is also able to perform some basic topological reachability checks on its own, thanks to the sophisticated API provided by Neo4J database. Notice that the reachability checks performed by Neo4JManager do not take into account nodes behaviour and their configurations, i.e. it only considers the raw topology and properties verified using the standard graph theory. Further complex checks are implemented by VeriGraph exploiting FOL formulas and VNF models, as described in previous sections.

Thanks to the collaboration of these two modules, we are able to check some reachability-derived invariants against the service graph under deployment, both from a topological perspective (in order to speed up the verification phase in case of bad service graph specification) and from a behavioural point of view of the involved VNFs. In particular, VeriGraph can check the violation of:

(i) *Reachability*: the tool builds a formal model of each middlebox involved in the path between source and destination and checks if at least one packet from the source node can arrive at the destination. In this case we are sure that if

[8] https://github.com/netgroup-polito/neo4jmanager

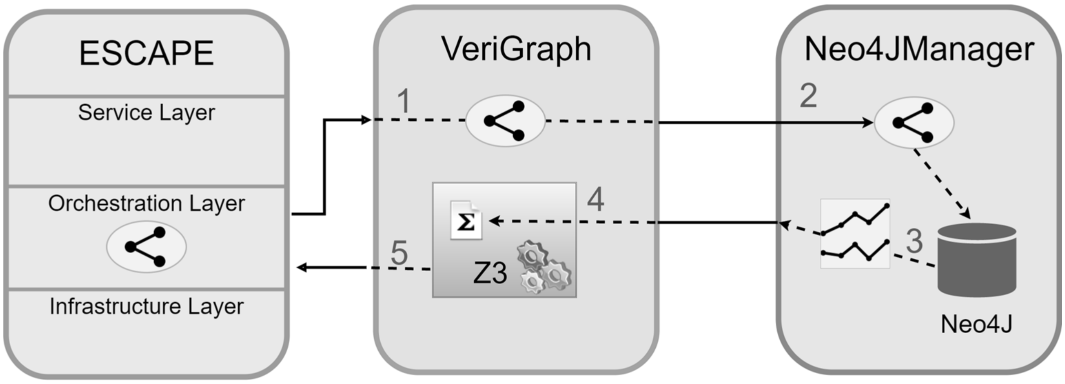

Fig. 3.6 VeriGraph design architecture.

the reachability property is not satisfied, there is no connection between the two nodes in the graph;

(ii) *Isolation*: the tool checks that no packet that goes from source to destination passes through a certain middlebox. In practice, this is equivalent to verifying that the source cannot reach this middlebox in all the paths toward the destination node;

(iii) *Traversal*: the tool checks that all packets that go from source to destination pass through a middlebox. This kind of invariants is the opposite of the aforementioned isolation properties, because we have to check both that the source reaches the middlebox in every paths towards the destination and that the middlebox reaches always the destination.

3.4.3 UNIFY pre-deployment verification: use case

In Section 3.4.1, we defined a possible scenario to validate our verification approach. The presented results were mainly obtained by feeding the Z3 SMT solver with the proper formulas to model the considered graph and the corresponding invariants. In

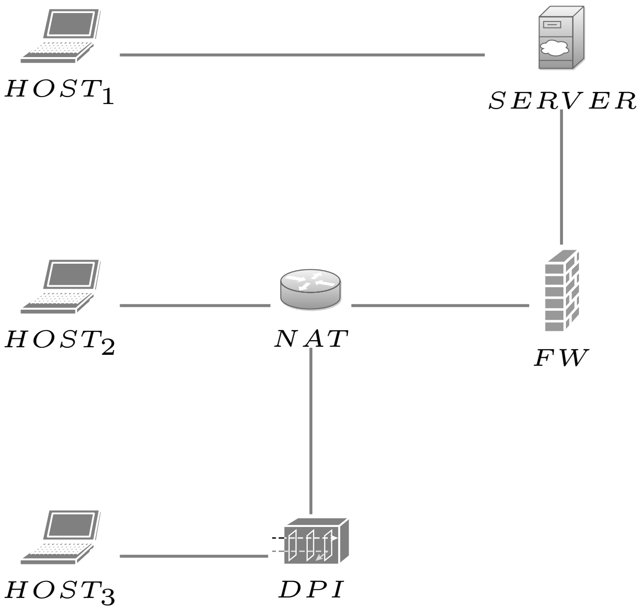

Fig. 3.7 Network Function-Forwarding Graph use case.

this section, we further refine our validation by considering the newly implemented components, i.e. VeriGraph with Neo4JManager and the complete tool workflow.

In particular, we are located into the Orchestration layer of the UNIFY architecture, and in turn, of ESCAPE. The workflow of VeriGraph is composed of several phases (shown in Figure 3.6): *(i)* receive a NF-FG and the invariants to check as input from the VNF architecture, e.g., ESCAPE (step 1 in Figure 3.6); *(ii)* store the graph and all of its nodes into Neo4JManager (step 2), by means of the VeriGraph REST API ("*NF-FG Creation*"); *(iii)* extract all the possible chains (step 3) to/from the nodes involved in the provided policies("*Chains Extraction*"); *(iv)* generate a verification scenario for each chain extracted (step 4), including all the necessary FOL formulas to model the encountered middleboxes ("*FOL Formula Creation*"); *(v)* run all the verification tasks generated in the previous step ("*Z3 running*") and return the verification result (step 5); *(vi)* raise an exception if any error occurred during the previous step.

The aim is to understand the impact of each phase of the verification on the overall deployment of the service graph. To do this, in Figure 3.7 we briefly report the network scenario we used to check if VeriGraph has been correctly integrated into the UNIFY architecture and in particular into Escape. Moreover, we evaluated the

Table 3.1 Verification times achieved by VeriGraph with a bad network configurations.

Reachability Property	Verification Result	NF-FG Creation + Chain Extraction	FOL Formula Creation	Z3 Running	Verification Time
Host1 ->Server	✓	169ms	87ms	54ms	310ms
Host2 ->Server	x	196ms	130ms	28ms	354ms
Host3 ->Server	x	175ms	88ms	30ms	293ms

Table 3.2 Verification times achieved by VeriGraph with a correct network configurations.

Reachability Property	Verification Result	NF-FG Creation + Chain Extraction	FOL Formula Creation	Z3 Running	Verification Time
Host1 ->Server	✓	188ms	94ms	61ms	343ms
Host2 ->Server	✓	209ms	86ms	65ms	355ms
Host3 ->Server	✓	153ms	97ms	107ms	357ms

time spent by VeriGraph in each of its phases. This evaluation has been performed under different network configurations on the same NF-FG.

In order to check if each host is able to reach the web server (shown in Figure 3.7), initially we have configured the firewall to drop all traffic coming from the NAT function. Table 3.1 shows in fact that the reachability verification between the endhost `Host2` and the web server `Server` is not satisfied (column "*Verification Result*"), similarly for the endhost `Host3` and the server.

We then fix the network behaviour by changing the firewall configuration and allowing traffic between any end host and the web server. As we expected, VeriGraph checks successfully the reachability invariants in the requested service graph for any end host.

From the point of view of a VNF Orchestrator, VeriGraph fulfils the aforementioned network verification constraints, which are the use of non-complex modelling approaches in order to achieve reasonable verification times and to run the service graph checks before of the deployment phase. The timing results shown in Tables 3.1 and 3.2 show that VeriGraph is able to check service graph request in less than 400ms, with different VNF configurations. This result does not impact the overall deployment step, because the achieved verification times are completely acceptable for humans who hardly notice this difference.

3.5 Future works: scalability issues

One desirable property of a verification tool is the scalability to enterprise and operator networks. Given the promising evaluation results achieved, we spent more efforts to the scalability issues in verifying complex service graphs.

Even though the boolean modelling-based approach seems to be more promising than traditional model checking techniques that generally suffer of memory consumption, VeriGraph must be able to treat also complex service graphs.

The initial approach adopted in VeriGraph is to model a VNF and the whole service graph as a set of First Order Logic (FOL) formulas (e.g., NAT model shown in Figure 3.3). Unfortunately, FOL is an undecidable logic due to the presence of existential and universal quantifiers (i.e., $\exists$ and $\forall$) and this would cause Z3 and other SMT solvers to timeout. Thus, a tentative for making VeriGraph able to treat also large-sized networks is to change the modelling strategies used to represent the VNF catalogue and the forwarding principles of the networks.

To ensure the formulas to describe a service graph (i.e., the union of the models of the network forwarding principle and of each VNF involved in the graph) are decidable, we plan to make the FOL-based formulas in a Skolemized normal form in order to remove the existential quantifiers.

In order to apply this new formalism, a possible strategy is to reformulate our assertions with variables that represent an *event*, which represents a particular condition in the network forwarding like the delivery of a packet. We can consider also the *cause* that has triggered an event, which is an event itself. To better understand, we can consider the receiving of a packet as an event which is caused by the packet's sending, which could be considered as an event as well.

In this reformulation of the FOL-models, each event has its own properties that characterize it like source and destination nodes, time, etc.... Those properties can be assigned to an event with predicate functions, which are pure FOL functions that do not have an a priory interpretation, but they allow any interpretation that is consistent with the constraints over the function itself. We can also use predicate functions to retrieve the cause of an event (i.e., by defining a "*cause*" function) and to check which kind of event we are considering. In particular, we can consider two kinds of event, that are the sending and receiving events (differentiated by the "*isSend*" and "*isRecv*" functions).

$$
\begin{aligned}
isRecv(event(n\_0,n\_1,p\_0,t\_0)) \implies\ & isSend(cause(event(n\_0,n\_1,p\_0,t\_0)))\wedge \\
& src(cause(event(n\_0,n\_1,p\_0,t\_0))) == n\_0 \wedge \\
& dest(cause(event(n\_0,n\_1,p\_0,t\_0))) == n\_0 \wedge \\
& pkt(cause(event(n\_0,n\_1,p\_0,t\_0))) == p\_0 \wedge \\
& time(cause(event(n\_0,n\_1,p\_0,t\_0))) < \\
& time(event(n\_0,n\_1,p\_0,t\_0))
\end{aligned}
$$

Fig. 3.8 Formula for modelling packets receiving and sending.

To complete the formula transformation, we have to replace the use of the quantifiers (i.e., ∀ and ∃) in the model formulas with equivalent formulas that contain Skolem functions instead of symbols.

To better understand how to model a network with this new approach, one aspect in the network forwarding that we can model is when a network node receives a packet. The action of receiving a packet implies that a network node (different from who is receiving) has sent that packet before. This means that in our model a receiving event is caused by a sending event, with some basic conditions: the source node of the cause must be equal to the source node of the receiving event (similar for the destination node); the sending and receiving events are delivering the same packets and the first event happens before than the second one. This concept can be expressed in the new formalism shown in Figure 3.8.

For what concerns VNF modelling, let us consider the firewall functionality shown in Figure 3.4 as example, which drops packets based on an Access Control List (ACL). As we have already mentioned, our ACL firewall will forward only those packets with source and destination addresses that do not match any entry in the ACL. Then, the formula 3.9 states that the event e of sending ($isSend(e)$) done by the firewall itself as source node ($src(e) == firewall$) implies that the cause of e was a receipt event ($isRecv(cause(e))$), whose destination node is the firewall ($dest(cause(e)) == firewall$). Formula 3.9 also indicates that the packet sent during the event e is the same packet received during the $cause(e)$ (i.e., $pkt(cause(e)) == pkt(e)$) and that the source and destination addresses inside this packet are not present in the ACL of the firewall ($\neg acl(src(pkt(e)), dest(pkt(e)))$).

$$
\begin{aligned}
(isSend(e) \wedge src(e) == firewall) \implies\ & isRecv(cause(e)) \wedge \\
& dest(cause(e)) == firewall \wedge \\
& pkt(cause(e)) == pkt(e) \wedge \\
& \neg acl(src(pkt(e)), dest(pkt(e)), \forall e
\end{aligned}
$$

Fig. 3.9 ACL firewall model in Skolemized form.

The issues related to the scalability of a verification approach are very critical tasks for a VNF Orchestrator, which can face with large-sized service graphs and needs a verification service able to treat all the incoming graph requests. Thus, we sustain the importance of improving VeriGraph in this direction and of making it able to scale *w.r.t.* the service graph size.

Chapter 4

A Proposal for Seamless Configuration of VNFs

In this part of the thesis, we now move to the presentation of the problems derived from the configuration of service graphs and possible solutions.

Traditional service provisioning models took a great deal of time and effort. The creation of chain-based network services in the past meant acquiring network devices and cabling them together in the required sequence. Each service required a specialized hardware device, and each device had to be individually configured with its own command syntax.

We already know that the introduction of NFV and SDN has improved the management and configuration of network services because, on one hand, moving network functions into virtual instances means that building SFCs no longer requires acquiring hardware, and on the other hand, moving network management out of the hardware and put it in a logical centralized appliance means the replacement of proprietary device configuration languages in favour of standard solutions and the increment of agility and flexibility in the network configuration.

What the literature has proposed during the last years are solutions for selecting the best VNFs to create the desired network service, for allocating resources for the selected VNFs (e.g., number of VNF instances or CPU/Operating System/-bandwidth/... requirements), for instantiating the network connections between the network functions.

All of these steps are needed to deploy a service graph, but what we still miss to complete this phase of the network service life cycle is a way to push down functional configurations into the VNF (e.g., blacklist for DNS filter or IP addresses for router interfaces) without using vendor-specific interfaces.

Enabling a flexible and seamless configuration process in the service life cycle allows providers to improve the quality of services offered to the final users, achieving many advantages. An example is the possibility of fixing bugs in case of verification failures. However, we can envision several challenges to address: *(i)* network functions may require different ways (REST API, CLI, SMTP, etc...) for being configured and *(ii)* the semantics of a configuration depend on the network function itself (e.g., router parameters are clearly different from firewall ones).

Thus in this chapter we investigate a possible solution for configuring VNFs compared with the other proposals in the research world.

4.1 Problem statement and contributions

In order to enable a user-centric service model, where final users can customize their service graphs by selecting network functions either among the natively supported catalog or provided by third-parties, and to simplify the management aspects, Data Center Providers (DCPs) rely on the use of data center management software, named cloud managers (CM), such as OpenStack[1]. A cloud manager is a software suite that handles management tasks such as the deployment of Virtual Machines (VM), the creation of the virtual network topology, and more.

Cloud managers can rely on additional systems for configuring network services and functions: examples are Puppet, Chef, Ansible and others. One advantage of such solutions is the possibility to integrate a network function without any modification to the function itself, thanks to the use of additional software modules such as agents or plug-ins. This approach, in fact, relieves VNF developers from the burden of developing a special instance of their functions for each particular CM adopted by the provider. However these tools miss a high-level agility in configuring VNFs because they are targeted to very expert users (i.e., providers, administrators, developers etc.). Existing configuration systems generally have a very steep learning

[1] https://www.openstack.org/

curve and this implies that non-expert tenants cannot build their virtual networks without learning how to use the tool allowed by their DCP. Moreover, configuration tools generally force users to create VNF configurations in the format needed by the tools themselves, without providing a high-level representation of such configuration parameters (e.g., no separation between the representation of an IP address and its real value). In this way, such tools cannot exploit the advantages provided by model-based configuration approaches, which have been recently proposed in the literature (e.g., [33]).

Model-based configuration means defining a representation (i.e., a data model) for the configuration parameters of each VNF. The configuration, defined through the above data model, is then automatically processed internally by the system, hence generating the actual configuration parameters (e.g., IP addresses associated to all the interfaces of a router VNF), which are then pushed into the VNF. An advantage of the separation of the actual VNF configuration from its high-level representation is that it does not force DCPs to use a single tool (e.g., Puppet) for configuring network functions. This makes VNF insertion simpler, because developers can integrate their VNF implementations in any CM that supports the same model-based approach.

Among the recent solutions that follow this trend, an informal working group of network operators has proposed the use of vendor-neutral data models through the OpenConfig project[2]. OpenConfig aims at creating a set of YANG-based models of network functions, leaving the choice of the strategies for pushing the actual configuration, automatically derived from the data model, to the operators such as the DCP. ForCES [33] is another example of VNF-independent configuration approach, which relies on a unified model of network abstractions and makes uses of a single protocol to control the VNF lifecycle. In order to use this approach with already existing VNFs, either the VNFs have to be updated with the addition of an implementation of the protocols specified by the above standard, or new adaptation components have to be provided, in order to guarantee the seamless integration of the existing VNFs in the new architecture.

In line with this recent trend, we propose a new architecture based on vendor-neutral network function data models defined through *VNF descriptions* that (*i*) facilitates the DCP in building a rich VNF catalog by adding services that can offer a simple and uniform configuration plane to tenants, (*ii*) enables non-expert tenants

[2] https://www.openconfig.net/

to configure their network services through that simple and coherent interface, and (*iii*) offers VNF developers an easy way to integrate their services in the CM used by the DCP.

Relying on previous works [34] and inspired by advantages of the existing agent/plug-in-based solutions that avoid changes in the VNF code, we design an architecture that exploits additional *translation modules*. The distinctive features of our additional modules are that they are VNF-agnostic (differently from configuration plug-ins), while they are specific per-configuration strategy. In this thesis, we define *configuration strategy* as the combination of the protocol used to send the configuration (e.g., SSH, NETCONF, SNMP, etc.) and configuration method (e.g., command line interface (CLI), REST API, file etc.) that has to be used to fully configure a VNF. In particular, the characteristic to transparently support multiple configuration strategies is a clear differentiation compared to existing solutions, which usually make use of a single protocol or interface (e.g., Puppet).

In summary, this work examines how DCPs can enhance their CM for enabling the proposed configuration approach. In particular, we propose a solution that relies on new components and a different way to input configuration data to perform all the necessary configuration tasks, and does neither require additional per-VNF control modules (such as an additional VM that provides the adaptation layer between the CM and the VNF, or an additional module running natively in the CM), nor changes in the VNF code to integrate it in the CM, such as the the implementation of an additional configuration strategy in the VNF itself.

4.2 Related work

In this section we investigate existing approaches for configuring VNFs. Even though the research world has presented different works somehow related to ours, most of the examined solutions focus on a subset of the problem we face, for this reason we have grouped them in several categories.

4.2.1 Agent-Based Configuration Approach

Several tools have been proposed to make the configuration and installation of additional software easy in a data center. Puppet[3], Chef[4], Ansible[5] are examples of existing configuration management systems, which aim at simplifying the task of managing large and complex compute deployments and keeping the system up to date.

This kind of solutions is based on a master-agent model (like Puppet), which requires the use of agents running in each node to configure and update the network services installed on it. One advantage derived from an agent-based approach is that the VNF developer does not take upon himself the overhead of managing the communication between the VNF and NFV architecture, because it is handled by the provider. This means that VNF developers do not have to implement additional software, apart from their functions, and also providers avoid the installation of unknown software (i.e., VNF-specific plug-ins) in their network.

On the other side, such solutions generally present drawbacks like steep learning curve, no abstraction of the network function configurations and also difficulty in managing physical instances of network functions due to the installation of non-native support agents (e.g., Puppet's agent). Moreover, most of them rely on a centralized management module, which has to collect the configurations of all the functions and services installed in the network and manage them, bearing all the well-known problems of a centralized solution. On the contrary, we envision a solution modular and logically distributed.

4.2.2 Protocol-Based Configuration Approach

Among the investigated solutions for configuring network functions, SNMP [35] and NETCONF [36] are two protocols that lay on data model languages. SNMP relies on SMIv2, while NETCONF on YANG, which has been indicated in [37] as a better data modelling language compared to other languages (e.g., XML schemas).

From a provider perspective, the use of a single configuration protocol, like SNMP or NETCONF, may limit the VNF catalog: all the non-SNMP/NETCONF

[3] https://puppetlabs.com/

[4] https://www.chef.io/

[5] http://www.ansible.com/

functions cannot be selected in case these are the only protocols supported by the provider. The use of a more flexible architecture that can enable more than one configuration strategy (e.g., NETCONF, RPC, REST, etc.) can relive, on one side, the VNF developers from integrating the support of further protocols in their functions, and, on the other side, make the provider free to chose the best configuration strategies among the available ones, based on, for example, security implications.

4.2.3 Model-based configuration approaches

A recent proposal in model-based configuration is represented by the open-source OpenDayLight (ODL)[6] SDN controller, which exploits a model-based service abstraction layer in order to create programmable (and configurable) network services. In particular, ODL enables the integration of third-parties VNFs thanks to the use of plug-ins in charge of the communication between the VNF and ODL, and YANG-based models that specify the data structures used and the messages supported by the northbound interface of the VNF itself.

At the best of our knowledge, the peculiarity of the approach taken by ODL does not consist in the model-driven abstraction, because many solutions encourage the use of models to represent data structures and primitives to generate the VNF configuration and push it down to the function itself. Instead, while the SDN controller needs VNF-specific plug-ins that are developed by the VNF developers and that have to implement an RPC API as northbound interface (i.e., toward the SDN controller), we envision the use of configuration modules that are VNF-agnostic and that can communicate with the VNF through any protocol or API (i.e., configuration strategy) supported by the provider.

The ForCES framework [38], defined by the IETF Forwarding and Control Element Separation working group, is another example of model-based configuration approach and it addresses the creation, configuration, and resource assignment of VNFs, exploiting an object-oriented model. In this context, [33] argues for the need of a unifying common network abstraction model for both forwarding aspects and network functions. This model is processed by a Network Function Manager in charge of accessing to each device through appropriate APIs and managing their life

[6] https://www.opendaylight.org/

cycle. The authors have also provided a proof-of-concept of the ForCES applicability in an NFV architecture [39].

However, instead of exploiting existing configuration strategies already supported by a network function, as it is envisioned by our work, their solution uses an additional protocol (namely ForCES) for configuration and management purposes. Moreover, the ForCES framework was designed for configuring network datapaths by means of an XML schema: on the contrary, recent trends define function models for configuration purposes, without considering network connections and resources and use YANG as data modelling language, instead of XML schemas, for the aforementioned reasons.

OpenConfig is an Industry-based working group that proposes another model-based approach for configuration and management and is currently building a public database of vendor-neutral data models of network functions, created using the YANG language. The development of these vendor-neutral data models can facilitate VNF integration, because, it focuses on making function model natively supported on networking hardware and software platforms reusable for any provider network. The main goal of the OpenConfig project is concentrated on the modeling phase, leaving providers free to implement any strategy for pushing configurations into network functions. Our main objective is instead to define all the features needed to handle all the steps of a complete configuration process.

4.2.4 Other Approaches

Finally, it is worth summarizing how some of the most recent orchestration architectures proposed in the literature refer to the VNF configuration problem, starting with the well-known ETSI-driven NFV architecture. Even though the ETSI NFV project has considered the problem of configuring VNFs and has proposed the Management and Orchestrator component (MANO) to take care of management and configuration tasks in the NFV architecture, only few works (e.g., [40]) are currently available that investigate this issue.

An example of MANO solution is vConductor, presented by Shen *et al.* [41], which enables users to define their virtual networks. The authors have designed their solution to exploit OpenStack network and compute resource provisioning frameworks, providing a proof of concept. In particular their prototype exploits a data model for service, computing and networking aspects, neglecting configuration parameters and data as we, instead, envision.

Juju[7], instead, is an open-source framework for modelling and deploying applications in service oriented architectures. It is seen more as a VNF Manager of the ETSI NFV architecture than a VNF Orchestrator, because it aims at simplifying and automating the service (or application) integration instead of implementing orchestration functionalities. Juju, in fact, is in charge of modeling services and tacking care of their delivery, regardless of the underlying infrastructure. Any other aspect of a service delivery is left to other tools, such as the service configuration that is managed in Juju thanks to the aforementioned Puppet and Chef. In this case we can find all of the pro and cons of the approaches adopted by such tools as discussed in Section 4.2.1.

Another important architecture has been developed within the FP7 project UNIFY [42], which aims at orchestrating any VNFs available in the whole network of the telecom operator, which addresses the configuration problem by defining a dedicated module as responsible of this task. A proof of concept prototype of this architecture is available through the ESCAPE framework [32], which deploys virtual networks by processing a data model (i.e., Network Function-Forwarding Graph - NF-FG). However, the problem of the configuration has not been properly investigated, as the NF-FG model includes only basic parameters such as IP addresses and cannot be seen as an acceptable answer to the very general problem of VNF (and service) configuration.

4.3 Objectives and Challenges

This chapter presents a flexible, scalable and VNF-agnostic solution to configure VNFs based on (*i*) a VNF-independent formalism to describe the data model of any VNF, (*ii*) a set of VNF-dependent high-level data models (based on the previous formalism) that describe the function itself, and (*iii*) a set of VNF-independent components called *translators* and *gateways* that are in charge of translating high-level configuration directives into the actual configuration commands, which are VNF-specific.

[7] https://jujucharms.com/

In our case, network functions (and their models) can be provided dynamically to be deployed in tenant virtual networks and CMs must guarantee their complete integration, even when these functions are not known in advance to the CM (e.g., in case of third-parties VNFs). This involves allowing the communication with other components (such as other network functions) and configuring the functions themselves. In particular the problem we faced is that after creating the virtual network, configuring the network paths (e.g., OpenFlow rules) and installing the chosen VNFs, tenants have to configure them, in terms of functional and behavioral information (e.g., black-listed domains for DNS filter).

The enhancements of CMs for enabling seamless configuration can bring benefits to all the actors involved, which are the DCP, VNF developers and Tenants.

From a DCP point of view, the use of a flexible and automated configuration approach can facilitate the insertion of new VNFs into its catalog, as the manual intervention of the DCP is no longer required each time a new VNF has to be added, with well-known consequences in terms of provisioning agility (minutes instead of weeks). This has a beneficial impact also on costs, as the adoption of VNF-independent high-level formalisms for data models reduces the difficulties in configuring different VNFs and favours the migration to industry standards such as the one proposed by the OpenConfig project, based on the YANG language.

Furthermore, a model-based approach enables also DCPs to enforce additional controls such as integrity checks and verification of the configuration correctness before actually deploying the requested virtual network, as proposed in some recent works [20, 24]. Another example is the implementation of an automatic reconfiguration process, such as the realignment of configuration parameters across multiple VNFs (e.g., the IP network assigned by the DHCP and the IP subnet used by the firewall to filter incoming traffic), although this requires the development of new advanced automated tools that guarantee the correctness of the generated configuration.

A possible answer to the above problem can be found in [40], which focuses on security applications and exploits refinement-based techniques to generate and deploy the proper functional configurations in the controlled VNFs. The resulting configurations are automatically derived from a set of high-level security statements, defined by the tenant itself, and are proved to be correct thanks to the mathematical background those techniques are built upon.

From a VNF developer perspective, instead, our solution would relieve developers from the burden of integrating their VNFs in every architecture, for example, which may require the development of CM-specific plug-ins. Hence a model-based approach can make VNFs immediately usable in any present and future DCP architectures (i.e., CMs), without the necessity of special integration efforts (e.g., reusing VNF models). In particular, our solution is designed so that VNFs can be integrated without supporting additional protocols, but exploiting those configuration strategies the function natively supports.

Finally, tenants can benefit from an enriched sets of functions, hence more services. Moreover, they are allowed to build and operate virtual networks without knowing the low-level configuration details (e.g., command line of a router or configuration files for a DNS filter) because our solution hides such technical details. In fact, DCPs could also provide a unified API (e.g., a dashboard) in order to facilitate tenants to experience a uniform way to program and configure the entire network infrastructure, including both topology information (e.g., links and VNFs) and the configuration required by the VNFs themselves. For example, a tenant could be able to configure a router through the same API that he used to deploy the function into the network.

While the advantages of having an architecture able to automatically configure VNFs according to the model-based approach are clear, we can envision two problems. First, the semantic of a configuration depends on the network function itself, as the parameters used to configure a router are clearly different from the ones needed by a firewall. This requires (*i*) VNF-specific data models, although created with a language that is VNF-independent and hence can support arbitrary functions, and (*ii*) a set of components able to dynamically understand such descriptions and apply them to the target VNF. Second, network functions may require different strategies for being configured: some support configuration methods like a web-based interface, a REST API, or an SSH-based configuration; others can be configured via SNMP, and more. This requires the translation of high-level configuration directives coming from the tenant into the specific commands available in the chosen configuration strategy (e.g., SNMP MIBs, configuration files) and the implementation of a communication protocol in charge of transferring the above configuration to the target VNF.

The target architecture should avoid the insertion of any VNF-specific configuration component inside the DCP's network, such as dedicated software modules that provide the interface between the uniform and user-friendly configuration interface exposed to the tenants and the actual configuration strategy supported by the specific VNF. In addition, the VNF images should be kept unchanged independently of the CM under consideration and the configuration strategies chosen by the DCP and/or supported by the VNFs. Finally, in case a VNF supports multiple configuration strategies, the architecture should be able to allow the DCP to choose the best one based on a cost function and/or its management policies.

4.4 The approach

This section presents first the high-level overview of the whole architecture, then it will show a more detailed view of the components in charge of configuring VNFs and the inputs they require for their tasks, and the respective actors that are in charge of that data.

4.4.1 Architecture overview

In order to perform a complete service deployment, including the configuration of the VNF, the CM needs both the VNF image and its data model, which is collected in a *VNF description* and must be provided by the VNF developer. The DCP has to store both the VNF image and description in the proper modules of its CM, in order to have them available when the tenant issues a service request (Figure 4.1). At this point, the tenant can configure its VNFs through a VNF-agnostic interface provided by the CM (e.g., REST API, dashboard, etc.). Finally, the configuration module will automatically generate and push the actual configuration in the VNFs, making the requested service fully operative.

In this process it is important that the VNF description is defined in a unified format in order to help mainly developers and DCPs. The VNF developer can define the main functional information (e.g., firewall policy) and the configuration protocols and methods (i.e., configuration strategies) for pushing them into the VNF in a way that is recognized by any DCP, which, in turn, are able to add to their catalog and use any VNF that adheres to the unified description format. In order for this to

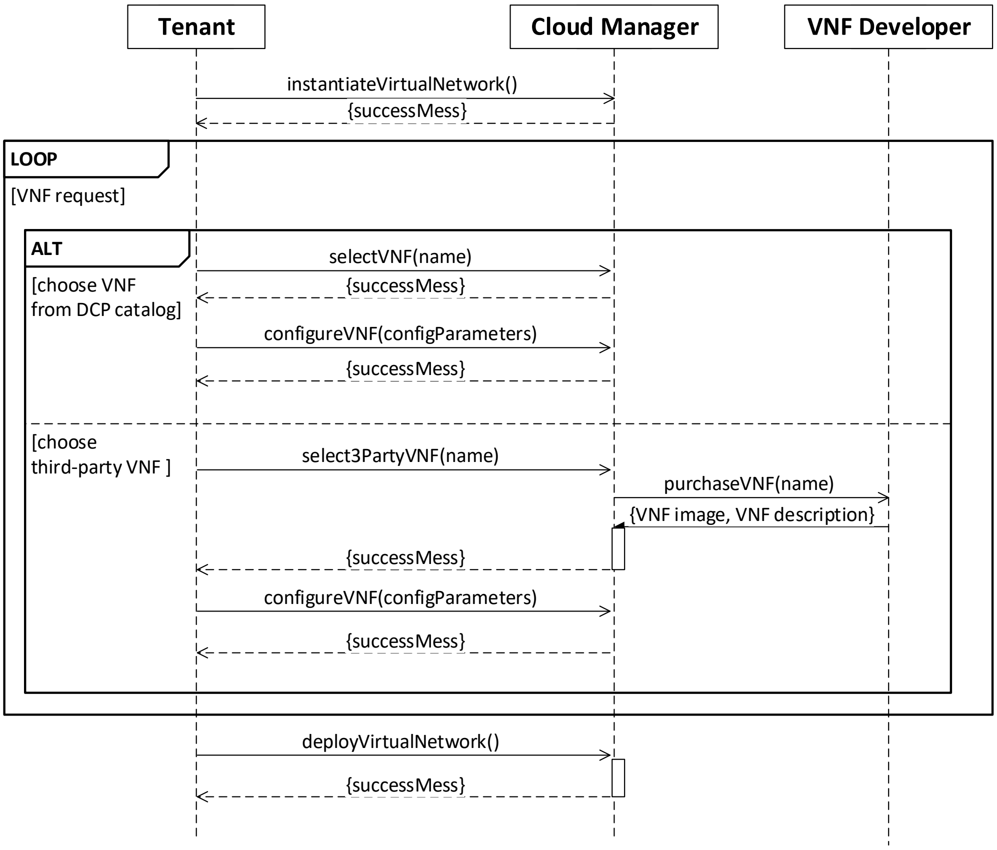

Fig. 4.1 Interaction between different actors.

be possible, DCPs must be able to configure any VNF regardless of their intrinsic details.

In order to enable this kind of configuration service, many features are needed in a CM, in particular the integration of modules that perform the configuration and the modification of the exposed interface to receive the VNF configuration parameters.

A possible configuration-oriented architecture of a CM is shown in Figure 4.2 (*case A*): a master component is in charge of generating the VNF configurations, while the agents are software modules installed in the network where the VNFs are deployed, that monitor if the VNFs need some configuration updates and, in case alert the master.

This is the approach adopted by solutions like Chef, Puppets, etc., where the master/agent communication is based on HTTP API. This type of solution avoids the

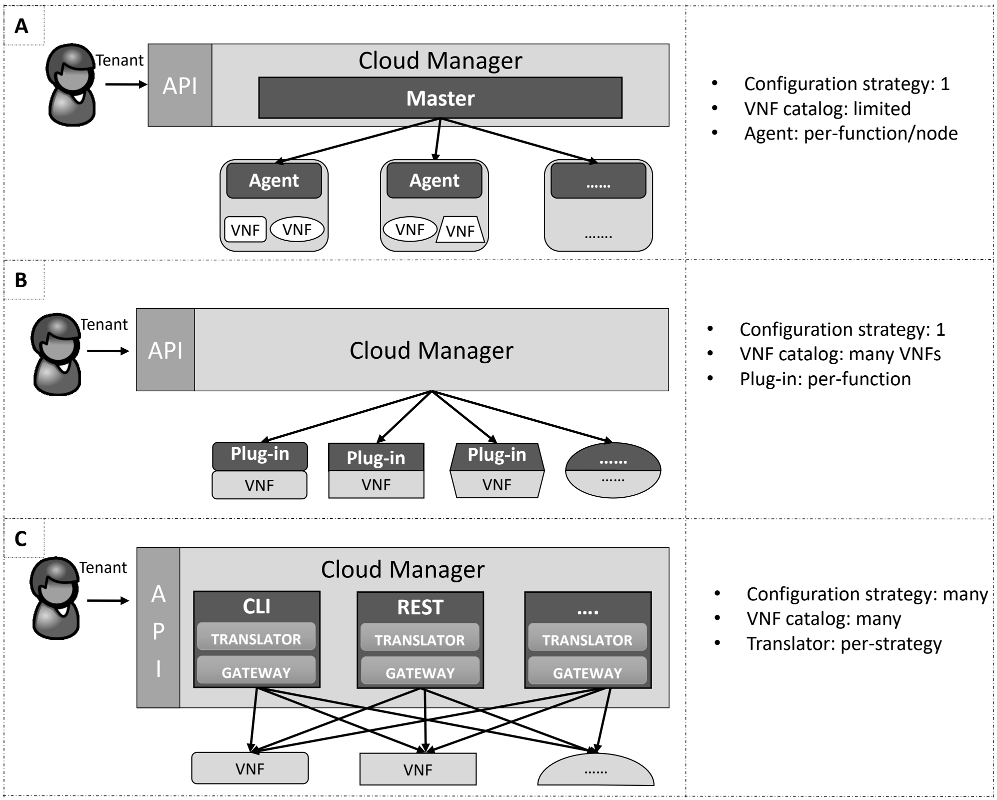

Fig. 4.2 Possible configuration-oriented CM architectures.

necessity to develop additional code (i.e., adaptation of the VNFs to the configuration protocol chosen by the DCP, or the implementation of VNF-specific plug-ins), hence facilitating the operations of the VNF developer. However, this architecture presents some limitations *(i)* with respect to the VNF catalog, because functions that cannot be configured through the agent cannot be integrated in the system (or further effort must be spent for their integration), and *(ii)* it requires the presence of an agent that runs aside each VNFs.

Another possible architecture is depicted in *case B* (Figure 4.2), where the configuration engine is not centralized inside the CM, but it is moved to VNF-specific plug-ins. This is the approach currently adopted by OpenDayLight, which requires a control plug-in developed by the VNF developer to enable the communication between VNF and cloud manager by means of, for example, an RPC API. From a DCP point of view, this solution leads to: *(i)* decreased VNF integration cost than *case A*; *(ii)* a richer set of offered virtual services thanks to the simplicity in integrating new VNFs; *(iii)* the necessity to execute control modules that are developed by third-parties, which might introduce additional security issues.

Our solution is represented by the architecture shown in *case C* (Figure 4.2), which translates the high-level configuration parameters in the actual VNF configuration commands and pushes them into the function. This solution (*i*) avoids any change in the VNF source code and/or the implementation of additional configuration plug-ins, (*ii*) avoids the installation of control agents in the network in order to better scale with the number of deployed VNFs and, finally, (*iii*) supports multiple configuration strategies (e.g., CLI, REST, etc.).

In particular, our approach is based on the splitting the configuration engine in two orthogonal (and sequential) tasks, which consists in the translation of high-level configuration parameters into a particular format required by a VNF and their delivery to the function. Thus, a *translator* takes care of the first task and a *gateway* will take care of the latter, transferring and installing the VNF configuration by using one of the configuration strategies already supported by the function itself.

This logically distributed architecture allows providers to optimize the configuration task by instantiating a variable number of translators and gateways, possibly only upon request, based on the current load of the system and the number of VNFs that have to be configured in order to implement the overall service request.

The use of translators and gateways allows the system to increase the number of supported configuration strategies (hence, VNFs that require unconventional configuration methods and protocols) without impacting on the existing ones, which continue to operate as usual. Moreover, these modules exploit the data model descriptions of the VNFs for which the configuration has to be created, enabling DCPs to support an unlimited number of network functions. Also the separation of translating VNF configuration from the task of delivering it to the VNF allows to split the inputs needed by the new components. In particular, translators and gateways will use different parts of the VNF description, which are: *VNF object model*, *translation rules* and *access parameters*. The next sections will describe in detail these inputs and how they are exploited by the new components.

4.4.2 Configuration translators

Since each configuration strategy has its own peculiarities (e.g., for a CLI-based configuration, it is necessary to know the commands for enabling administrative authorization), the architecture includes a *configuration translator* for each configuration strategy the DCP wants to support (Figure 4.3). A translator hence must be aware of all the particular techniques and quirks needed for the strategy it is in charge of.

The use of separated translators makes also the system more extensible and manageable, as it allows an easier insertion, replacement and removal of supported configuration strategies: when the DCP wants to support a new strategy, he has just to make a new translator available (and, in turn, a new gateway). Furthermore, the system becomes scalable with respect to the number of VNFs running in the system: one translator (and gateway) enables the integration of a number of network functions that support that strategy. The larger the number of VNFs, the larger the number of translators and gateways that are instantiated. This solution contrasts with the limited scalability of agent-based architectures where a single agent may become a bottleneck.

Translator inputs. In order to perform its job, a translator needs the *VNF Object Model* (OM), which is the VNF-specific model that we exploit to represent the function data inside the system (*point 1* in Figure 4.3). In particular a VNF object model represents a description of the data-structure instantiated for storing the configuration parameters of a VNF. This means that each VNF must be associated with its object model in order to be correctly integrated into the system. Note that more than one OM may be necessary for the same type of VNF. For example, a firewall from a first manufacturer can support features that are different from the ones supported by a firewall of another manufacturer, requiring further configuration parameters for such additional features and hence a different object model.

Moreover since the object model is only the specification of the data structure used to store configuration parameters, an instance of the OM for each deployed VNF is stored inside the CM, namely the *VNF Object Model instance* (OM instance). For instance, we can consider the VNF OM as a class declaration in an object-oriented programming language, while the VNF OM instance can be seen as a particular instance of that class. In particular, when the CM receives the request to deploy a new set of services (*point 2*), for each VNF that is being instantiated a specific OM instance is created (*point 3*). Referring to the previous example, if the tenant has required two VNFs of the same firewall, two OM instances are created from the object model of that particular firewall, one for each firewall VNF deployed in the

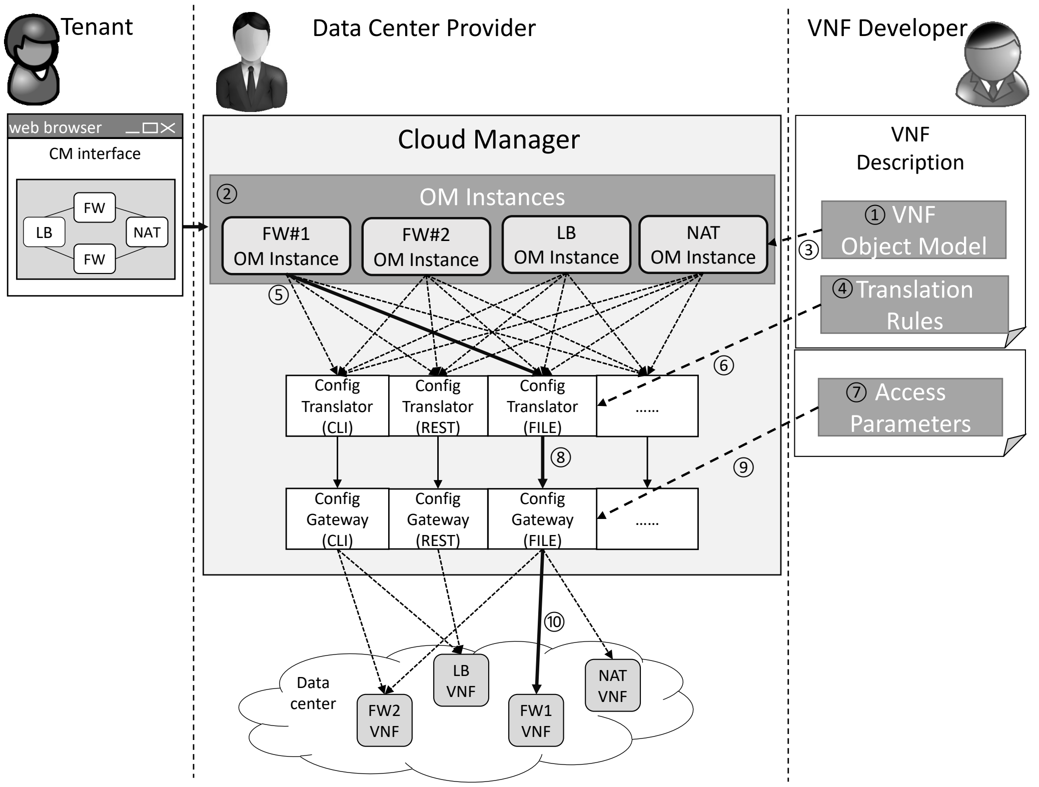

Fig. 4.3 Overview of a CM architecture for configuring VNFs.

network. Each OM instance will contain the set of policy rules configured for its associated firewall.

Among the other aforementioned advantages, the use of data models makes any changes in DCP-provided API easier and transparent for the internal processes of the system. This avoids also the use of data-structure formats for collecting VNF configurations that would be translator-specific.

Another input coming from the CM is a set of *translation rules* (*point 4*), i.e. directives used to drive the translator in generating the configuration of the VNF in the right format. They express the way to translate the structure and content of the OM instance into the specific structure/format required by the VNF. Referring to the previous example of the firewall, translation rules specify the format of policy rules according to the specific firewall in use.

The particular format used to configure a VNF depends also on the configuration strategy supported by the function itself (e.g., CLI, REST API, file etc.). For instance, a configuration through REST interface has certainly a different format from one used for CLI-based configuration.

To better clarify the idea under the translation rules, let us consider the previous example of the firewall and suppose that a policy rule can be set through a command line like "*add rule -source 130.192.31.24 -destination 8.8.8.8 -action ACCEPT*": an example of translation rule may be like "*add rule -source IP_VALUE -destination IP_VALUE -action ACTION_VALUE*", where the actual configuration parameters values (i.e., IP address and action) are stored in the OM instance of that firewall. Further details about the format of the translation rules in our solution are presented later.

As well as the object model, translation rules are both configuration strategy- and VNF-specific, since each network function has its own primitives to be used in the configuration phase. Hence both the VNF object model and translation rules can be provided by the developer through the VNF description.

With respect to which inputs the configuration translators need, the new modules have been designed to receive: *(i)* configuration parameters saved into the OM instance of the VNF (*point 5* in Figure 4.3); *(ii)* translation rules for deploying such parameters into the VNF in the right format (*point 6*).

It is interesting to note that an OM instance is self-descriptive and hence translators can discover the structure of an OM from any instance of that model.

4.4.3 Configuration gateways

The proposed architecture includes also a *configuration gateway* for each translator and, in turn, for each strategy supported (Figure 4.3). Gateways are in charge of delivering the actual VNF configuration into the function by means of the configuration strategy for which it is authorized. In order to achieve this goal, a gateway needs surely the result of the configuration translation process, which represents the final configuration of the function (*point 8* in Figure 4.3). However a configuration gateway requires another input to perform its goal, namely the *access parameters*.

Gateway input. The *access parameters* (*point 7* in Figure 4.3) are used to instruct the system on how to contact the VNF and update its configuration, in order to complete the configuration of the VNF itself. Examples of access parameters are IP addresses, port numbers, administrative credentials, commands for entering in configuration mode and everything that describes how to add the policy rules inside the firewall we have considered before.

In our vision access parameters should be standardized for each configuration strategy, because each strategy needs different information: for example, in a configuration though files, CMs must know the path where configuration files are stored. DCPs and developers can then set the actual values of those parameters: DCPs would establish the management information internal to his architecture (e.g., IP addresses of control interfaces), while developers would set parameters related to the internal mechanism of the VNF (e.g., root credentials). This implies that access parameters are strategy-specific, but some of them are also VNF-independent. This is the reason why we do not include such parameters into VNF descriptions. Further details on how access parameters are stored in a real implementation of the architecture are provided later.

4.5 Implementation and evaluation

This section describes a proof-of-concept implementation of the proposed architecture. We start by presenting some details that have not been included in the previous description in order to keep the architecture description more generic (i.e., inputs format and languages). We then continue with a description of our prototype.

4.5.1 Object Model

The VNF Object Model is based on the YANG data modeling language [43], developed by IETF and extended for our purposes. YANG has been designed to model configuration data and state, which can be manipulated through a protocol such as NETCONF. YANG was chosen because of it is protocol-agnostic, implementation-independent and human-readable. YANG is also easy to extend with new directives without impacting the compatibility with previous implementations. Furthermore it is oriented to network configuration tasks, hence it provides an excellent foundation for our problem as well.

This language offers also a wide set of directives to validate its statements. Examples are type checking, default values, mandatory/optional statements and their

```
<schema>
  <element name="router">
    <complexType>
    ....
      <sequence>
        <element name="interfaces" minOccurs="1"
                          maxOccurs="unbounded">
          <attribute name="name" type="string"/>
          <complexType>
            <sequence>
              <element name="ethernet" minOccurs="1"
                          maxOccurs="unbounded">
                <complexType>
                  <attribute name="name" type="string"/>
                  <attribute name="address" type="tns:ipv4"/>
                  <attribute name="hwid" type="tns:eth"/>
                </complexType>
              </element>
            </sequence>
          </complexType>
          <key name="nameEthernetKey">
            <selector xpath="ethernet"/>
            <field xpath="name"/>
          </key>
        </element>
      </sequence>
    </complexType>
    <key name="nameInterfaceKey">
      <selector xpath="interfaces"/>
      <field xpath="name"/>
    </key>
  </element>
</schema>
```

Listing 4.1 XML Schema language example: an excerpt of a router VNF description.

cardinality, value ranges checking, and other. While other simple validations are possible through the definition of new YANG types, more complex validations (e.g., dependency checking between statements) would require new extensions. The support for validating primitives could allow VNF developers to include directives that can be checked against configuration parameters provided by the tenant. However we are interested in the configuration process and prefer to leave the checking and verification of configuration correctness as future work. Hence, in our implementation YANG has been exploited to define the VNF object model, which includes the most significant data structures that are required to properly configure the function.

While the YANG language provides the set of advantages listed before, our architecture can be implemented with any other language that present similar characteristics; a possible alternative to YANG is represented by the XML Schema. In this respect, XML Schema is more mature and already well standardized, but it is more verbose, as shown by comparing the same data structure defined in XML Schema (Listing 4.1), where we have defined the same data structure shown in Listing 4.2. In addition, YANG is being adopted by different projects in the field of network management such as OpenConfig, and new software artifacts such as the OpenDaylight SDN controller, hence it should be more familiar at lest to the network managers.

In the example of a possible YANG Object Model (i.e., VNF description) shown in Figure 4.2, we have define a structure to describe the state of the Ethernet interfaces of a router. The idea is to have a data structure to enumerate all the interfaces of a router (the top-level `interfaces` list) and, for each of them, store all their network and physical addresses[8] (respectively the leafs `address` and `hwid` in the nested `ethernet` list).

In our solution, the YANG VNF description file includes also translation rules (presented in the next section), which are VNF-specific.

4.5.2 Translation rules

As shown in Listing 4.2, translation rules take the form of special comments in the YANG-based VNF description, using the following structure:

```
//ConfigTransl:<Transl_N>:<Rule_N> <Rule_V>
```

where <`Transl_N`> specifies which configuration translator (and, in turn, configuration strategy) the rule belongs to and can assume values like “`file`”, “`cli`”, “`rest`”, etc.. Instead, <`Rule_N`> and <`Rule_V`> represent the rule name and value, interpreted as strings. This structure allows to group all the rules for a given translator under a specific prefix, in a way that is similar to the concept of the *namespace*. This permits the presence of multiple translation rules in the same YANG file, which can be useful when the VNF can support different configuration strategies.

[8] Usually a network interface is assigned only one network and physical address, but this is not true in the general case.

```
module router {

import ietf-inet-types { prefix inet; }
import ietf-yang-types { prefix yang; }

...

list interfaces {

 //ConfigTransl:file:header "//Start Interface List\n";}
 //ConfigTransl:file:list_format "%NAME {\n";
 //ConfigTransl:file:separators "\n}\n";
 //ConfigTransl:file:footer "}\n//End Interface List";

 key name;
 leaf name {
   type string;
   }

 list ethernet {

   //ConfigTransl:file:list_format "%NAME %VALUE {\n";
   //ConfigTransl:file:separators "\n";
   //ConfigTransl:file:footer "}\n";

   key name;
   leaf name {
     type string;
     }

   leaf address {
     //ConfigTransl:file:leaf_format "%NAME %VALUE\n";
     type inet:ipv4-address;
     }

   leaf hwid {
     //ConfigTransl:file:leaf_format "hw-id %VALUE\n";
     type yang:mac-address;
     }
   }
 }
}
```

Listing 4.2 YANG language example: an excerpt of a router VNF description.

The example presented in Listing 4.2, which assumes that the router is configured through a file-based translator, shows some translation rules that create the properly formatted output, which are: *(i)* `header` and `footer` are inserted respectively before and after the current YANG element (e.g., `list` or `leaf`) when generating the final configuration; *(ii)* `separators` is used to divide child nodes of the current statement; *(iii)* `list_format` and `leaf_format` work like a `printf` of the C language, in which `%NAME` and `%VALUE` are expanded with values depending on the context. In particular, `%NAME` and `%VALUE` represent respectively the name of their YANG statement (e.g., “ethernet" for the `list ethernet` and “address" for the `leaf address`) and its actual value (in the case of a `list`, it will be the value of its key).

Although other configuration strategies may need additional (or different) information such as the exact ordering sequence of the commands to be issued in a CLI-based configuration, this does not represent a problem, as new translation rules can be defined with the format needed by the specific translator. Furthermore, we could leverage hierarchical data structures, which are natively offered by YANG. For instance, the current implementation serializes the YANG Object Model of a VNF, hence assigning a lower priority to the nested elements than their root statement.

None of the keywords is mandatory: an extreme case, thus, is a YANG statement that does not have any translation rule. In this case that node will not appear in the configuration output.

4.5.3 Access parameters

In general, access parameter are VNF-independent, but depend on the specific configuration strategy chosen by the VNF (e.g., a network-based configuration requires the TCP port to connect to, while a file-based configuration requires to know where that file is located). Hence the DCP has to define the proper set of access parameters for each supported configuration strategy. This is the reason why in our solution the above parameters are not included in the VNF description, but they are stored in another object that is used only by the CM and may not be fully exported to the tenant. In order to simplify the deployment, also access parameters are described using the YANG language.

Moreover, a new OM instance for the access parameters is automatically created when a VNF is deployed and associated to the function, because this instance must

```
module ConfigTransl2File {
  list access_param {
    key name;
    leaf name {
      type string;
      }
    leaf ip_address {
      type string;
      }
    leaf port {
      type string;
      }
    leaf user_name{
      type string;
      }
    leaf user_key {
      type string;
      }
    leaf commands {
      type string;
      }
    leaf file_name {
      type string;
      }
    leaf file_path {
      type string;
      }
  }
}
```

Listing 4.3 Excerpt of access parameter object model.

store the actual values of access parameters for loading a new configuration into that VNF. The access parameter OM instance is also associated to its function thanks to the `name` field (Listing 4.3), which contains the VNF identifier inside the system.

An example of OM of access parameters for file-based configuration is shown in Listing 4.3. Here VNF developers, even tenants, must be able to set the access parameters related to the VNF only. In other words, they must not have the privileges for setting parameters like IP addresses of management interfaces and others. An example of possible OM instance associated with the aforementioned model and related to a router VNF is shown in Listing 4.4, written in a JSON-like format.

The choice of the YANG language for describing the access parameters was taken also because most of the parameters we need to describe are simple (like

```
{
  "access_param":
  {
    "name": "Router_94",
    "ip_address": "130.192.31.94",
    "port": "2001",
    "user_name": "router_admin",
    "user_key": "admin",
    "commands": "load /configuration/config.boot",
    "file_name": "config.boot",
    "file_path": "/configuration"
  }
}
```

Listing 4.4 Possible content of an access parameter OM instance.

IPv4/IPv6 address, configuration file name and path, etc.) and natively supported by YANG. In any case, if needed, we can leverage the additional YANG types defined by IETF in [44].

Finally to recap the configuration process, Figure 4.4 shows a detailed view of the whole architecture including all the inputs. The VNF developer has provided both the VNF image (*point 0*) to launch the function instance, and the other inputs required by the system. In particular, the VNF object model (*point 1*) allows to build automatically the CM interface (dotted line) and OM instance associated to that function (dashed line). Translation rules are instead sent to the translators (*point 2*), while access parameters (*point 3*) are stored into a gateway-specific OM instance (its structure is defined by the DCP, as mentioned before). The OM instance associated to the access parameters is automatically created when a new VNF is deployed: this instance is then associated to the VNF and used by the gateway later on.

Through the CM interface (e.g., the DCP web dashboard in Figure 4.4), tenants can set the VNF configuration parameters (*point 4*), which are stored into the OM instance of the function (*point 5*). After that, this instance (*point 6*) is passed to the translator selected based on the preferred configuration strategy (i.e., CLI-based translator in the example). The combination of this input and the translation rules is used to generate the actual VNF configuration (*point 7*). Finally, the configuration gateway can retrieve both access parameters (*point 9*) and the produced configuration (i.e., translator output - point *8*) to complete the VNF configuration process (*point 10*).

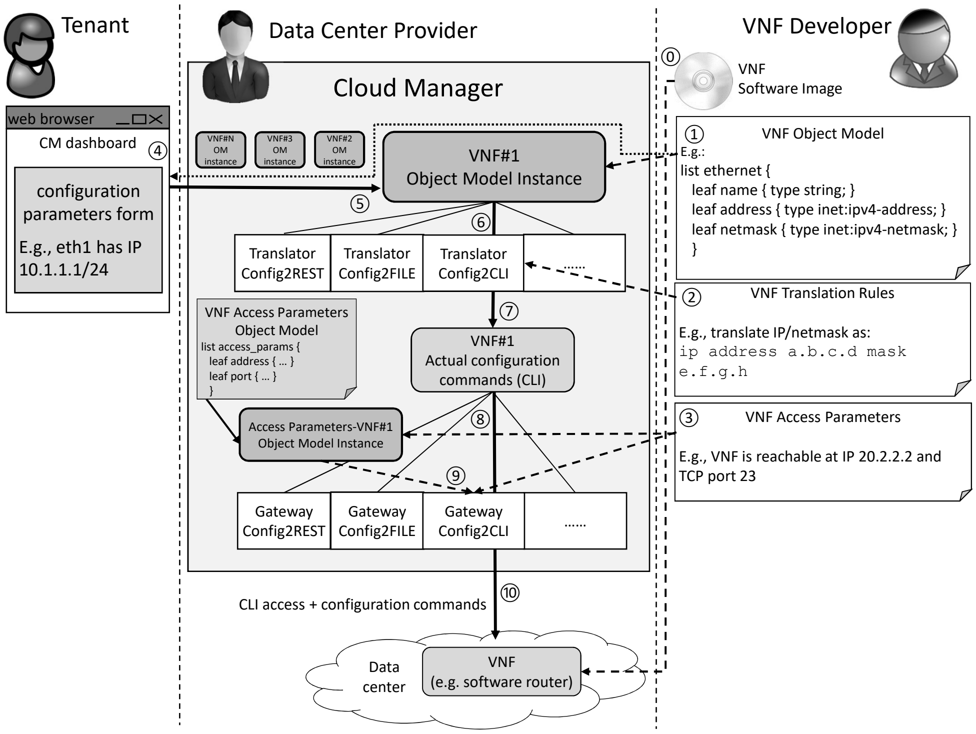
Tenant
Data Center Provider
VNF Developer
Cloud Manager
web browser
CM dashboard
④
configuration parameters form
E.g., eth1 has IP 10.1.1.1/24
VNF#N OM instance
VNF#3 OM instance
VNF#2 OM instance
VNF#1 Object Model Instance
⑤
⑥
Translator Config2REST
Translator Config2FILE
Translator Config2CLI
......
⑦
VNF Access Parameters Object Model
list access_params {
leaf address { ... }
leaf port { ... }
}
VNF#1 Actual configuration commands (CLI)
Access Parameters-VNF#1 Object Model Instance
⑧
⑨
Gateway Config2REST
Gateway Config2FILE
Gateway Config2CLI
......
⑩
CLI access + configuration commands
Data center
VNF (e.g. software router)
⓪
VNF Software Image
① VNF Object Model
E.g.:
list ethernet {
leaf name { type string; }
leaf address { type inet:ipv4-address; }
leaf netmask { type inet:ipv4-netmask; }
}
② VNF Translation Rules
E.g., translate IP/netmask as:
ip address a.b.c.d mask e.f.g.h
③ VNF Access Parameters
E.g., VNF is reachable at IP 20.2.2.2 and TCP port 23

Fig. 4.4 Detailed overview of the enhancements in CM architecture.

4.5.4 ConfigTransl2File Prototype

Having defined the language and formats of the additional inputs required by the new components, we have also implemented a prototype for validating and testing the effectiveness of our solution. In our prototype, a C++ library, namely `ConfigTranslLib`, has been designed to support several configuration strategies. We have implemented a translator/gateway prototype, namely `ConfigTransl2File`, to configure VNFs by means of files, regardless of their format (e.g., XML, text or more).

This translator receives inputs through a REST interface exposed by the CM, which are: (*i*) YANG OM instances of VNFs, where the translator can retrieve the configuration parameters chosen by tenant and the configuration file structure required by the function; (*ii*) translation rules, which have been stored in the VNF OM instance as well as the configuration parameters.

For the sake of simplicity, in our implementation the whole set of access parameters is configurable through the REST API. However, in a real vendor's implementa-

```
module bind9 {

list zone {

  //ConfigTransl:file:list_format "%NAME \"%VALUE\" {\n";
  //ConfigTransl:file:separators ";\n";
  //ConfigTransl:file:footer "};\n ";

  key name;
  leaf name {
    type string;
    }

  leaf type {
    //ConfigTransl:file:leaf_format "%NAME %VALUE\n";
    type string;
    }

  leaf file {
    //ConfigTransl:file:leaf_format "%NAME \" %VALUE\"\n";
    type string;
    }

  leaf master {
    //ConfigTransl:file:leaf_format "%NAME { %VALUE; };\n";
    type string;
    }
  }
}
```

Listing 4.5 An excerpt of the Bind9 YANG description file.

tion, just a subset of those parameters must be exposed and made public to tenants and developers. The access parameters are then stored into another OM instance, specific for the `ConfigTransl2File` translator.

Finally it is worth noting that our solution is able to support VNFs that could require multiple configuration files. The `ConfigTransl2File` library can be instructed to write different portions of the same YANG OM into different configuration files, so that VNFs that require it can dump different parts of their data into different locations. This can be done because of the object model abstraction: the root element of a YANG module, for example a YANG list, has no difference from a nested YANG statement (e.g., container, leaf-list, list) under it, then these two elements of an object model can be the entry points associated to different configuration files of the same function.

```
zone ‘‘example.com" {
  type slave;
  file ‘‘db.example.com";
  masters { 192.168.1.10; };
}
```

Listing 4.6 Excerpt of the generated Bind9 configuration file.

4.5.5 Validation

We validated our architecture by implementing the components required to configure two VNFs, Bind9 and Vyatta Core, respectively a DNS server and a software router, which represent two well-known, albeit very different, network functions. We decided to benchmark the performance of our prototype using one VNF at a time, omitting the case in which multiple VNFs are deployed (hence, need to be configured) at the same time. In fact, our current proof-of-concept prototype handles the two VNFs sequentially, hence requiring a total time for the configuration that is the sum of the individual components. However, it is trivial to implement the architecture with multiple gateways and translators, all running in parallel, hence achieving a configuration time that is independent from the number of VNFs that have to be configured.

Starting with the validation phase, in particular concerning the DNS server, we have defined the YANG-based description for Bind9. An excerpt is shown in Listing 4.5, while Listing 4.6 is the corresponding part of the Bind9 configuration file, generated by our prototype. As shown in Listing 4.6, we have configured Bind9 to act as Secondary Master (i.e., it gets the zone data from the Primary Master for that zone). Our validation methodology consists in sending configuration requests to our CM through its REST interface that aim at setting the Bind9 configuration parameters; the call triggers the `ConfigTransl2File` translator, which generates the above-mentioned configuration file. Another REST call is then issued to initialize the Bind9 access parameters, which are stored in the proper OM instance.

Having all of the required inputs, the system is able to push the final configuration file into the VNF and restart it. The successful deployment of the configuration is validated by interrogating the Bind9 instance and checking that the returned answer are coherent with the desired configuration.

A similar validation has been performed also for the second VNF, which involves the Vyatta Core router. An excerpt of its YANG description file is shown in Listing 4.2. In this case we have checked that the Vyatta instance is actually configured with the desired data by checking the reachability of the IP addresses on the interfaces and its static routes. Listing 4.7 shows an excerpt of the Vyatta configuration file that was correctly generated by the ConfigTransl2File prototype from the description shown in Listing 4.2.

```
//Start Interface List
...
interfaces {
  ethernet eth0 {
    address 130.192.31.94
    duplex auto
    hw-id 00:0c:29:64:66:1c
    mtu 1500
    smp_affinity auto
    speed auto
  }
}
...
//End Interface List
```

Listing 4.7 Excerpt of the Vyatta configuration file.

4.5.6 Testing results

Two metrics have been considered for evaluating the effectiveness of the proposed solution, which are (*i*) the elapsed time for generating configuration files and (*ii*) the reduction of complexity from a tenant prospective, which can be translated in the size (i.e., verbosity) of the generated file compared to the corresponding YANG source.

Starting with the first metric, Figure 4.5 plots the required time for generating the configuration file versus the size of such file. We have performed multiple test runs (i.e., about 100 executions per file dimension) for both Vyatta Core (square points in figure) and Bind9 (circle points).

As show in the graphs, we have obtained satisfactory trends, because, as we expect, the time required by ConfigTransl2File grows proportionally to the size of the configuration file. This means that our solution is able to handle configurations

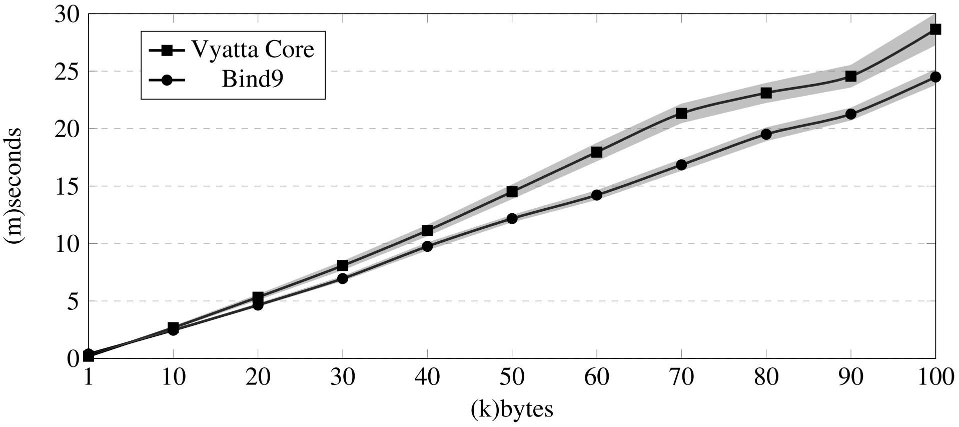

Fig. 4.5 Elapsed time for generating Vyatta Core and Bind9 configuration files, with 95% confidential intervals.

with a growing complexity, without requiring an exponential time increase. This is in line with the constraints of assuring a good experience to tenants. In addition to the size of a configuration file, other metrics may be considered to evaluate the complexity of a VNF configuration. The evaluation of many metrics is very useful from a provider point of view, because it can help in selecting the most suitable configuration method/API among the ones supported by a function. A metric that is the most suitable to evaluate a configuration strategy, in fact, may not be appropriate for another strategy (e.g., command priority may not be a relevant complexity indicator for file-based configuration). We leave the evaluation of our approach with other metrics as future work.

In addition, our tests demonstrate that, for real case scenarios, the time required to obtain the configuration file is on the order of tens of milliseconds. This result is also in line with the configuration times that are achieved with other agent-based solutions. In particular, we have identified Ansible, as one possible solution used to compare our approach. Ansible is based on agents that exploit the SSH protocol for their interaction with the VNF, which is usually supported by most VNFs. Furthermore, similarly to our approach, it avoids the installation of new agents in the VNF, hence preserving the original network function image. Compared to the configuration time needed by Ansible to push a configuration in both Bind9 and Vyatta, our solution performs slightly slower, but it never exceeds 40% the value obtained by Ansible, which is completely acceptable for humans who hardly notice this difference.

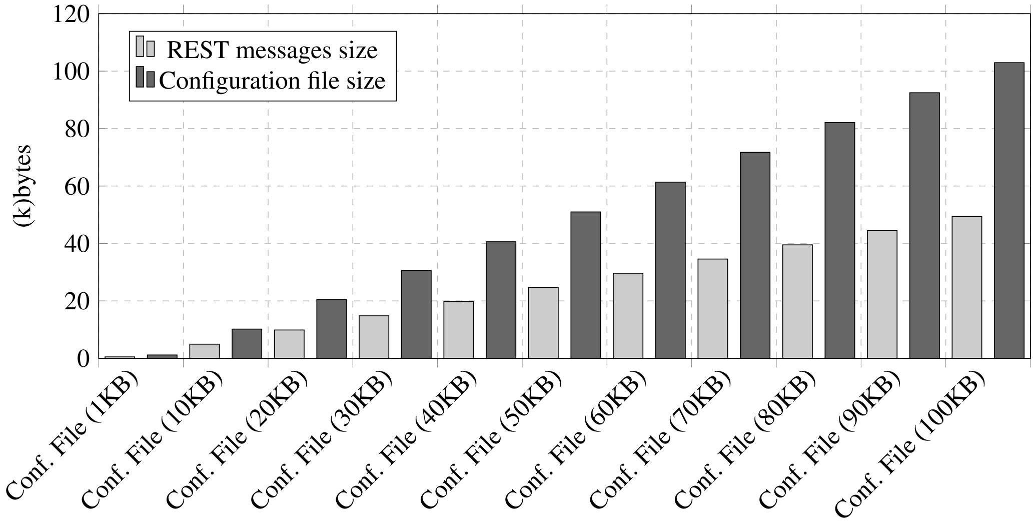

Fig. 4.6 Bind9 use case: reduction of configuration complexity.

Furthermore this result has been obtained with proof-of-concept code, which can be optimized in the future.

The two graphs, shown in Figure 4.5, report the 95% confidential interval and show that our solution takes less than 30ms in average in both use cases. The configuration time achieved by our prototype has a negligible impact on the total deployment time, which is usually on the order of tens of seconds when virtual machines have to be started. Hence, these results demonstrate that the introduction of our solution in the CM does not increase the service provisioning time experienced by tenants.

The second test suite aims at evaluating the reduction of complexity in configuring networks from a tenant prospective, achieved thanks to our prototype. In particular our solution allow the creation of YANG files in which only the main configuration parameters are exported, such as policy rules in a firewall VNF, avoiding all the details required by the specific configuration method and the possible syntactical rules (and keywords) required by the VNF native configuration method (e.g., firewall rule format, priority commands, special directives etc.).

As a metric to measure the complexity of the configuration, we used the size (in bytes) of the configuration files generated by our tool (dark grey bars in Figures 4.6 and 4.7), which represent the complexity of the native configuration method of the VNF. The above value has been compared with the size of the configuration

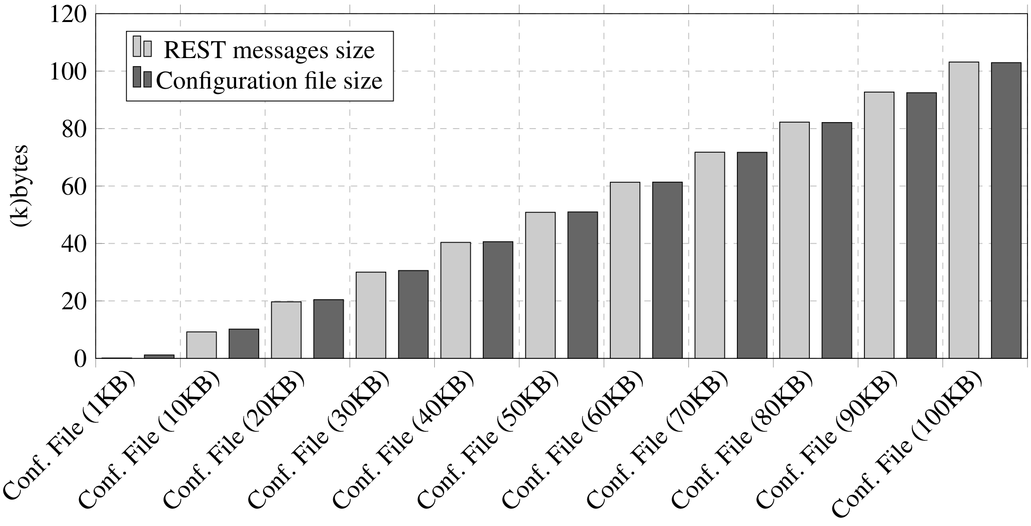

Fig. 4.7 Vyatta use case: reduction of configuration complexity.

messages that have been generated to push the configuration in the CM through our configuration REST APIs, which can be seen as the size of the same configuration with our approach (light grey bars in Figures 4.6 and 4.7). In particular, Figure 4.6 shows the results achieved in the Bind9 case, while Figure 4.7 depicts the case of the Vyatta Core router.

In the Bind9 case, the reduction of complexity, that is the difference between the size of the configuration file and REST messages, grows linearly, suggesting that tenants are facilitated in configuration phase. In the Vyatta case, the size of the configuration file is approximately equal to the REST message size, hence suggesting a similar complexity. However, it is worth noting that, with our approach, tenants are relieved from the burden of having a deep knowledge of how to configure their VNFs, since they must interact only with the CM, using a uniform configuration model across all VNFs. In addition, this result must not be attributed to a possible inefficiency of our solution as it is due to the differences existing between the two VNFs: each VNF implementation has its own configuration peculiarities and one function can be simpler than other in configuration phase. This result highlights the importance of supporting multiple configuration strategies and, in turn, multiple translators/gateways. For example, a configuration through the CLI may reduce significantly the configuration complexity of the Vyatta case, which means that the difference between the REST message payloads and the produced configuration is more evident than the configuration through file. Supporting many translators, the

DCP may select the most suitable one for configuring a specific VNF instance, based on their internal management policies and costs.

Concluding, our solution reduces the effort spent by tenants in configuring their virtual services, with a negligible impact in terms of configuration time and, likely, with a simplified configuration interface.

Chapter 5

Conclusion

It is argued that in the future Telecommunications infrastructures are likely to become highly dynamic, flexible and programmable production environments capable of providing any ICT services. Current OSS/BSS do not seem to cope with the requirements posed by this evolution: in fact, future operations will involve the management and control of a myriad of software processes, rather than closed physical nodes.

Today most providers offer dynamic network services, where users can achieve a certain degree of flexibility to cope with their needs. Users can build their own virtual services, creating network connections and deploying their preferred VNFs/services. Providers do not want to pose limitations also on VNF selection (i.e, they would like to allow users to chose VNFs either from the catalogue offered by providers or implemented by third-parties) and are looking for ways to enable further network services (e.g., network verification).

In this thesis, we focus on different aspects of a network service life cycle: we started from the anomaly analysis and reachability-checking in the chain-based services to move toward the functional configuration of the under-deployment service graph.

In particular, we started by presenting a solution to specify and verify the presence of anomalies in a forwarding policy, before the policy rules are enforced by the SDN Controller and installed into the network switches to create the desired service graph. The proposed approach enables precise and unambiguous specifications of policies and of related anomalies by using standard notations such as First Order logic and Horn clauses. Through the application of already existing verification engines, the approach allows a rigorous verification of the absence of anomalies and the consequent guarantee that the verified set of policies is anomaly-free.

This work also fulfils the challenges of network verification in a SDN/NFV context. We recall that providers need verification services with non complex modelling approaches that are scalable and fast in order to be applied before deploying the service graph. Our model, in fact, implements an early-verification approach since it is able to check medium- and large-sized networks in reasonable verification times from a VNF Orchestrator perspective.

Moreover, we achieve also high-level flexibility in network verification, because providers can define their own sets of anomalies, checking a pre-defined set of anomalies in every kind of network topology (e.g., bad policy rule specification or forwarding loops) and then assuring a minimum level of correctness. This approach could also become a wider, and more ambitious contribution. Since policy-based systems are largely widespread in data protection, filtering, access control, and many other policy domains, a useful contribution can be to extend this verification model in order to encompass different policy domains. An extended verification model could verify that a domain-specific policy is consistent also in the presence of policies belonging to other domains.

For what concerns reachability analysis in service graphs, we presented our initial contribution related to the verification process, which is one of the most important pillars in the SP-DevOps feedback cycle and a key enabler to support envisioned changes in the way providers deploy and operate new network services.

After generalizing the applicability of a state of the art approach to the verification of complex service graphs, we presented and discussed a couple of models we developed to validate our key ideas. Given the promising evaluation results achieved, we will address more efforts to some open topics in the VNF verification area such as scalability issues in verifying complex service graphs.

Other aspects can be investigated to address the limitations of the current approaches to check service graphs. For example, one limitation of the existing approaches is their possible low exploitation in real system deployments due to the complexity of the adopted modelling techniques (e.g., FOL in Z3). These are quite far from the traditional programming languages and paradigms developers know and use, with consequent high “barriers to entry”.

Other solutions are moving in this direction like Symnet [45], which actually offers a more friendly imperative language for expressing the data plane processing operations. However, this is still far from the programming languages that are part of the common background of VNF developers. In this sense, for example, the definition of a Java-like modelling language might be of great importance for a real exploitation of these verification techniques.

A second limitation of all current verification techniques is that they are not well integrated into the orchestration process, but they act as a post-processing step after the orchestration process. In this way, if errors are detected, service deployment fails because the orchestrator does not have clues about how to fix errors.

A possible progress beyond the current state of the art will be the development of formal approaches that, while providing final assurance levels similar to the ones of the state-of-the-art modelling and verification techniques, are incorporated into the service orchestration process, which in this way produces network configurations that, once deployed into the underlying infrastructure, are formally guaranteed to satisfy the required policies.

Finally, we have investigated the weaknesses of the current Cloud Managers used by providers in order to offer a solution for configuring the network functions using a simple and uniform configuration method, without at the same time forcing the provider to deploy additional per-VNF software modules or the VNF developer to adapt its code to the configuration tools chosen by the provider.

We have proposed a model-based approach to solve the above problem, which enables to configure VNFs in terms of functional parameters (e.g., IP address for a router and policy rules for a firewall), bringing multiple advantages for all the actors involved (i.e., provider, VNF developer and tenant). The cost and complexity reduction of integrating further VNFs is an example of a possible advantage for the provider and the VNF developer. While from the tenant perspective, our solution does not impact the service-provisioning time and simplifies the tenants interaction with the provider, because tenants are relieved from the burden of having a deep knowledge of how to configure each of their functions.

Possible future extensions could include further services provided by DCPs for verifying the correctness of the configuration generated and validating the correct integration of the desired configuration in the NFV architecture.

References

[1] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, feb 2013.

[2] Joel Halpern and Carlos Pignataro. Service function chaining (sfc) architecture. RFC 7665, RFC Editor, oct 2015. http://www.rfc-editor.org/rfc/rfc7665.txt.

[3] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, mar 2008.

[4] Joel M. Halpern and Carlos Pignataro. Service function chaining (sfc) architecture. RFC 7665, RFC Editor, oct 2015. http://www.ietf.org/rfc/rfc7665.txt.

[5] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT '14)*, pages 213–226. ACM, nov 2014.

[6] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. FatTire: declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN 2013)*, pages 109–114. ACM, aug 2013.

[7] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL, apr 2013. USENIX.

[8] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration (SafeConfig 2010)*, pages 37–44. ACM, oct 2010.

[9] Roberto Bifulco and Fabian Schneider. Openflow rules interactions: definition and detection. In *Proceedings of the IEEE Conference on SDN forthe Future Networks and Services (SDN4FNS 2013)*, pages 1–6. IEEE, nov 2013.

[10] Ehab S Al-Shaer and Hazem H Hamed. Modeling and Management of Firewall Policies. *IEEE Transactions on Network and Service Management*, 1(1):2–10, apr 2004.

[11] Bruno Lopes Alcantara Batista, Gustavo Augusto Lima de Campos, and Marcial P Fernandez. Flow-based conflict detection in openflow networks using first-order logic. In *Proceedings of the IEEE Symposium on Computers and Communication (ISCC 2014)*, pages 1–6. IEEE, mar 2014.

[12] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[13] Michal Bali. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing Ltd, 2009.

[14] Nicolae Paladi. Towards secure sdn policy management. In *Proceedings of the IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC 2015)*, pages 607–611. IEEE, dec 2015.

[15] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A nice way to test openflow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, pages 10–10. USENIX Association, apr 2012.

[16] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Philip Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, oct 2011.

[17] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–112, Lombard, IL, apr 2013. USENIX.

[18] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, apr 2012. USENIX.

[19] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the network with merlin. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotSDN 2013)*, page 24. ACM, nov 2013.

[20] Serena Spinoso, Matteo Virgilio, Wolfgang John, Antonio Manzalini, Guido Marchetto, and Riccardo Sisto. Formal verification of virtual network function graphs in an sp-devops context. In *Proceedings of the 4th European Conference on Service Oriented and Cloud Computing (ESOCC 2015)*, pages 253–262. Springer International Publishing, sep 2015.

[21] Matteo Virgilio. *Study and analysis of innovative network protocols and architectures*. PhD thesis, Politecnico di Torino, 2016.

[22] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *Proceedings of the 1th ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN '12)*, pages 121–126. ACM, aug 2012.

[23] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip A. Porras, and Guofei Gu. Model checking invariant security properties in openflow. In *Proceedings of IEEE International Conference on Communications ICC*, pages 1974–1979. IEEE, jun 2013.

[24] Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. Verifying isolation properties in the presence of middleboxes. *CoRR*, abs/1409.7687, 2014.

[25] Sanjeev Sharma and Bernie Coyne. *DevOps For Dummies*. Limited IBM Edition' book, oct 2013.

[26] Wolfgang John and Catalin Meirosu. Unify deliverable d4.1: Initial requirements for the sp-devops concept, universal node capabilities and proposed tools, 2014. https://www.fp7-unify.eu/index.php/results.html#Deliverables.

[27] Wolfgang John, Konstantinos Pentikousis, George Agapiou, Eduardo Jacob, Mario Kind, Antonio Manzalini, Fulvio Risso, Dimitri Staessens, Rebecca Steinert, and Catalin Meirosu. Research directions in network service chaining. In *Proceedings of the IEEE Conference on SDN for the Future Networks and Services (SDN4FNS 2013)*, pages 1–7. IEEE, nov 2013.

[28] Catalin Meirosu. Unify deliverable m4.1: Sp-devops concept evolution and initial plans for prototyping, 2014. https://www.fp7-unify.eu/index.php/results.html#Deliverables.

[29] Raj Jain and Subharthi Paul. Network virtualization and software defined networking for cloud computing: a survey. *Communications Magazine, IEEE*, 51(11):24–31, nov 2013.

[30] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer-Verlag, mar 2008.

[31] Per Danielsson, Jan Ekman, András Gulyás, Per Kreuger, Shaoteng Liu, Guido Marchetto, Felicián Németh, Bertrand Pechenot, István Pelle, Sachin Sharma, Riccardo Sisto, Pontus Sköldström, Serena Spinoso, Rebecca Steinert, and Matteo Virgilio. Unify deliverable d4.4: Public devopspro code base, 2016. https://www.fp7-unify.eu/index.php/results.html#Deliverables.

[32] Attila Csoma, Balázs Sonkoly, Levente Csikor, Felicián Németh, Andràs Gulyas, Wouter Tavernier, and Sahel Sahhaf. Escape: Extensible service chain prototyping environment using mininet, click, netconf and pox. *ACM SIGCOMM Computer Communication Review*, 44(4):125–126, aug 2014.

[33] Evangelos Haleplidis, Jamal Hadi Salim, Spyros Denazis, and Odysseas Koufopavlou. Towards a network abstraction model for SDN. *Journal of Network and Systems Management*, 23(2):309–327, jul 2015.

[34] Serena Spinoso, Marco Leogrande, Fulvio Risso, Sushil Singh, and Riccardo Sisto. Automatic configuration of opaque network functions in CMS. In *Proceedings of the IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14)*, pages 750–755. IEEE, dec 2014.

[35] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. A Simple Network Management Protocol (SNMP). RFC 6241, RFC Editor, may 1990. http://www.ietf.org/rfc/rfc1157.txt.

[36] Rob Enns, Martin Bjorklund, Jürgen Schöenwäelder, and Andy Ed. Bierman. Network Configuration Protocol (NETCONF). RFC 6241, RFC Editor, jun 2011. http://www.rfc-editor.org/rfc/rfc6241.txt.

[37] Hui Xu and Debao Xiao. Data modeling for netconf-based network management: Xml schema or yang. In *Proceedings of the 11th IEEE International Conference on Communication Technology (ICCT 2008)*, pages 561–564. IEEE, nov 2008.

[38] Lily Yang, Ram Dantu, T Anderson, and Ram Gopal. Forwarding and Control Element Separation (ForCES) Framework. RFC 3746, RFC Editor, apr 204. http://www.rfc-editor.org/rfc/rfc3746.txt.

[39] Evangelos Haleplidis, Spyros Denazis, Odysseas Koufopavlou, Diego Lopez, Damascene Joachimpillai, J Martin, Jamal Hadi Salim, and Kostas Pentikousis. ForCES applicability to SDN-enhanced NFV. In *Proceedings of the 3rd European Workshop on Software Defined Networks (EWSDN 2014)*, pages 43–48. IEEE, sep 2014.

[40] Cataldo Basile, Antonio Lioy, Christin Pitscheider, Fulvio Valenza, and Marco Vallini. A novel approach for integrating security policy enforcement with dynamic network virtualization. In *Proceedings of the 1st IEEE Conference on Network Softwarization (NetSoft 2015)*, pages 1–5. IEEE, April 2015.

[41] Wenyu Shen, Masahiro Yoshida, Kenji Minato, and Wataru Imajuku. vConductor: An enabler for achieving virtual network integration as a service. *IEEE Communications Magazine*, 53(2):116–124, feb 2015.

[42] András Császár, Wolfgang John, Mario Kind, Catalin Meirosu, Gergely Pongrácz, Dimitri Staessens, Attila Takács, and Fritz-Joachim Westphal. Unifying Cloud and Carrier Network: EU FP7 Project UNIFY. In *Proceedings of the 6th*

IEEE/ACM International Conference on Utility and Cloud Computing (UCC '13), pages 452–457. IEEE, dec 2013.

[43] Martin Bjorklund. YANG - A data modeling language for the Network Configuration Protocol (NETCONF). RFC 6020, RFC Editor, oct 2010. http://www.rfc-editor.org/rfc/rfc6020.txt.

[44] Jürgen Schoenwaelder. Common YANG Data Type. RFC 6991, RFC Editor, jul 2013. http://www.rfc-editor.org/rfc/rfc6991.txt.

[45] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: scalable symbolic execution for modern networks. In *Proceedings of the ACM Conference on SIGCOMM (SIGCOMM '16)*, pages 314–327. ACM, aug 2016.