Static Validation of Network Device Configurations in Virtualized Multi-tenant Datacenters

Yosuke Himura
Hitachi Yokohama Research Laboratory

Yoshiko Yasuda
Hitachi Yokohama Research Laboratory

***Abstract*—A recent fashion to reduce the device cost of datacenter networks is to leverage multi-tenancy, with which multiple virtual networks of different customers (tenants) are consolidated over a single physical infrastructure. The use of multi-tenancy, however, poses significant operational cost due to the complicated device configuration and the rise in risks of misconfigurations in such shared environment; Hence deploying a tenant requires network engineers to pay careful attention to validating the correctness of corresponding device configuration commands before actual deployment. In this work, we present a methodology that statically validates network device configurations before deployment by predicting a posteriori network states after execution of those commands. We mainly make use of graph-based techniques to achieve both (a) extensibility to deal with various types of network devices in datacenters (provided from abstracting configurations as graphs), as well as (b) interpretability of the validation results (provided from visualization – a key benefit from graphs). The evaluation with a dataset synthesized from a set of actual configuration files results in 68% reduction of the time to validate them on average.**

***Keywords*—network management, datacenter network, multi-tenancy, virtual network, network device configuration, configuration validation.**

I. Introduction

The use of multi-tenancy is one of the recent fashion in datacenter networking as a cost-effective method, with which a set of computing environments for multiple customers, called *tenants*, are provided by means of consolidation in a single shared physical infrastructure [1], [2]. Tens or even hundreds of tenants can be consolidated over a single infrastructure thanks to the recent advancement in processing performance and yet they are logically-independent each other due to various network virtualization technologies such as Virtual LAN (VLAN), virtual interface (VIF), and virtual router (VR). Diverse forms of tenant networks are created with the combination of these virtualization technologies, according to diverse requirements for enterprises (e.g., multi-tier system).

In spite of its cost-effectiveness, the deployment of tenants is still mostly based on manual configuration, which is notoriously time-consuming and prone to human errors. According to our observation [3], the basic workflow of deploying a tenant is as follows: (1) designing the network topology and functionality for a customer, (2) creating corresponding configuration settings written as command lines for devices, (3) validating the correctness of those setting commands to eliminate human errors, and (4) inputting those commands into network devices. Consolidating multiple virtual networks in a single physical infrastructure leads to complicated configuration, resulting in making those processes quite harder.

Validating device configuration commands has become particularly important, stemming from the fact that misconfigurations for a certain tenant in such shared environment will possibly affect or disrupt the status of the other tenants, which does not occur in conventional (i.e., single-tenant) networks. For instance, incorrect parameters such as wrong VLAN IDs may result in unwanted connection between different tenants, causing security violations, and/or unwanted disconnection inside a tenant network, leading to the incapability of communication among servers of the tenant. Network engineers hence have to spend much time and effort on validating configuration commands to eliminate human errors before actually deploying the tenant, i.e., before actually executing the crafted configuration commands for network devices.

In this work, we aim to develop a methodology to accelerate the process of before-deployment validation of network device configuration in multi-tenant datacenter networking. Specifically, this methodology statically examines a set of *new configuration commands* that are crafted by engineers to deploy a tenant (to be validated); It also analyzes a set of *current configuration files* obtained from current (i.e., before-deployment) status of network devices. By predicting the a posteriori network state after executing those commands (inferred from those two types of inputs), it automatically identifies the correctness of the new commands to help engineers detect incorrect ones.

We consider that such configuration validation should be both *extensible* as well as *interpretable* towards practical use for actual network operators. Here, the extensibility means the applicability of validation process to various types of networking devices including firewalls, load balancers, VPN appliances, and so on, in addition to conventional switches/routers. On the other hand, the interpretability represents the easiness of understanding validation results (e.g., which commands are incorrect and why), which is important to efficiently confirm the results and modify incorrect commands.

Regarding the validation of network configurations, there have been prior works developed in other contexts such as campus or ISP networks; However, a crucial difficulty in achieving both the extensibility and interpretability should be derived from the trade-off between the two as faced by those works. A few works have abstracted the validation issues with their original algebraic systems [4], [5] designed for extensibility to various contexts, at the sacrifice of interpretability caused by their significantly high-level abstraction. In contrast, other works have defined a number of detailed rules that configurations of specific devices must follow (e.g., BGP policy settings [6], [7]); Such concrete rules can specifically

978-3-901882-50-0

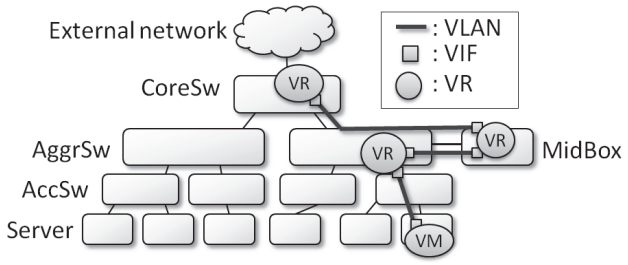

Fig. 1. Schematic representation of a conventional physical network of datacenter deploying a virtual network for a tenant.

point out the sources of errors (hence users can understand which command is wrong), but this approach has to focus only on certain specific cases, leading to less extensibility. Furthermore, as more crucial issues for those prior works of both the two approaches, those works mostly deal with configuration files of a certain snapshot of device state – not new configuration commands that have be validated before being input into devices, even though those works have mentioned the possibility of this. Hence, simple modification of prior works cannot be directly applied to our problem domain.

Instead, our basic approach is to leverage graphs, composed of vertices and edges, with which we detect violation of several general rules that any multi-tenant datacenter networking context has to follow. Configuration commands (e.g., creation/removal of VRs) are abstracted as graph operations (e.g., addition/deletion of vertices/edges), and those operations are validated based on those rules translated as a graph context. Also, validation results are visualized as graph pictures, with which network engineers can intuitively interpret the incorrect commands. This abstraction hence will hold the extensibility as well as the interpretability. We design and implement a validation methodology based on this approach, and evaluate it with datasets synthesized from configuration files actually used in a production datacenter. The main contribution of the present work is the original methodology of statically validating network device commands before actual deployment.

The reminder of this paper is organized as follows. We first present the basics of multi-tenant datacenter and difficulty in validating network device configuration to deploy a tenant (Sec. II). We discuss the primary approach to overcome this difficulty (Sec. III), and design a methodology to achieve this (Sec. IV), the effectiveness of which is evaluated with a dataset based on actual configuration files used in a business datacenter (Sec. V). We provide a discussion (Sec. VI), the current state of related works (Sec. VII), and a conclusion (Sec. VIII).

II. Background and motivation

In this section, we describe the form of physical infrastructure and virtual networks for tenants (Sec. II-A), how to configure network devices to deploy a tenant network (Sec. II-B), and the problem of difficulty in validating network configurations before deployment (Sec. II-C). The first two sections have already been mentioned in our previous work [3], but we briefly revisit them to make this paper self-contained.

TABLE I. Type of devices (shown in Figure 1).

Type	Description
Access switch (AccSw)	connects multiple servers basically with L2 functionalities
Aggregation switch (AggrSw)	aggregates access switches and connect middleboxes, providing L3 connectivity
Middle-box (MidBox)	provides network services such as firewall filtering, load balancing, NAT, VPN termination, or SSL acceleration
Core switch (CoreSw)	merges traffic from aggregation switches (if necessary) and connect to external networks (e.g., enterprise networks, or the Internet)

TABLE II. Type of virtualization technologies (shown in Figure 1).

Type	Description
Virtual Router (VR)	VR is logical instance for routing functionality inside a device. The Virtual Routing and Forwarding (VRF) is a common realization.
Virtual Interface (VIF)	VIF is logical instance representing network interface. Well-known realizations are VLAN interface and sub-interface.
Virtual LAN (VLAN)	VLAN is a conventional technology for L2 separation, which virtually separates a physical link into logically independent links (mainly IEEE 802.1Q)

A. Multi-tenancy datacenter network

Figure 1 represents a typical datacenter network. This figure shows that basic datacenters are composed of several types of network devices, which are detailed in Table I [1].

A single physical infrastructure consolidates separated virtual networks for multiple customers, which we define as *tenant networks*. Figure 1 also illustrates a tenant network deployed over the above-mentioned physical devices. Table II shows various types of virtualization technologies, which provide logical separation among tenants. Such virtualization technologies generate components of virtual network (i.e., routers, interfaces, or links – we name them *virtual resources*), and hence a tenant network can be interpreted as a collection of virtual resources.

B. Tenant configuration

Configuration settings in the networking literature are generally stored as configuration files, each of which is a list of commands of networking functions. Figure 2 represents two examples of configuration files used to construct a tenant network. Here, Figure 2(a) and (b) correspond to a list of configuration commands for a firewall as MidBox role and a switch as AggrSw role, respectively. Those configuration files are related to the tenant network depicted in Figure 1 [2].

These configuration files can be broken down in terms of virtual resources as follows.

- Allocating VR instances and specifying static routes (lines 1-4, 11-13): The virtual routers named vrA and v1 are allocated in the firewall and switch, respectively.

[1]Datacenters are usually redundantly constructed to achieve high availability, although we omit this from the figure for readability.

[2]These configuration files are partially modified for brevity and not exactly same as those of existing products.

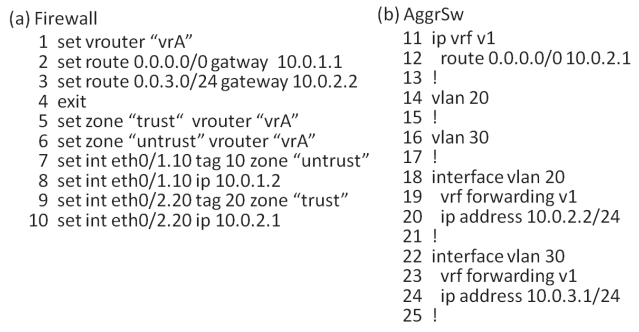

Fig. 2. Configuration commands of a firewall MidBox (left) and an AggrSw (right) for the tenant network depicted in Figure 1.

- Creating VIF instances and attaching them to a VR instance (lines 5-10, 18-25): For instance, the firewall creates two VIFs named eth0/1.10 and eth0/2.20, and connect these interfaces with the virtual router vrA via the definition of zones, each of which is a logical grouping of interfaces. Similarly, the switch creates VIFs 10 and 20, then connects them to the VR tagged as v1. These lines include the assignment of IP address on the interfaces.
- Defining VLAN instances and relating them to VIFs (lines 7, 9, 14-17, 18, 22): Some lines can be interpreted as creating VLAN instances from the viewpoint of virtual resources, although they are not explicitly declared. For example, lines 7 and 9 can be interpreted as creating VLANs 10 and 20 and attaching them to the physical interfaces eth0/1.10 and eth0/2.20.

Network engineers have to deal with such many parameters. Similar forms of configuration commands are detailed by a few commercial guides [1], [2]. Our empirical observation on actual datacenters is that network devices consists of thousands (or even tens of thousands) of lines of configuration commands to consolidate a number of tenants.

C. Problem in validating configurations

Network engineers create such configuration files to deploy a tenant, but validating the correctness of those configuration files is especially crucial because any single error among various types of parameters (Sec.II-B) can possibly cause degradation or ultimately failure of services. The use of multi-tenancy in datacenters has made such configuration validation more difficult. In addition to the need for domain-specific knowledge about manipulating various types of network devices, consolidation of multiple tenant over a single physical infrastructure leads to the significant complexity in device configurations; Readers should imagine configuration commands listed in Figure 2, the amount of which is further multiplied by the number of tenants (e.g., tens or even a hundred) but still they are written in low-level device-specific languages without any tenant context.

We consider that configuration validation should be both *extensible* and *interpretable* towards practical use, as we mentioned in Sec. I, but there is a crucial trade-off between the two as faced by the recent state-of-the-arts. Highly-abstracted mathematical systems are capable of dealing with diverse networking contexts, resulting in the lack of interpretability (which is crucial for practitioners, who are generally unfamiliar with abstract algebra), whereas defining a number of detailed rules of specific protocols/devices leads to the difficulty in handling diverse types of devices used in datacenters.

III. Approach: Graph-based Configuration Validation with Principle-based Rules

To handle the problem of difficulty in configuration validation, we select to use graphs, composed of vertices and edges, different from past works. Configuration commands are abstracted as graph operations (Sec. III-B), which are capable of handling various devices as well as capable of visualization to provide interpretability. As validation rules, we set several requirements that must generally be complied in multi-tenant datacenter contexts (Sec. III-C).

A. Input data

As the inputs for validation, we select to use *new configuration commands* crafted by engineers (which have to be validated before executed) and *current configuration files* obtained from current state of network devices (i.e., before deployment), which are used for sources of validation. The primary motivation of this choice is three-fold: (a) These files are generally managed by network engineers and hence can be used as an input for examination; (b) The pre-defined syntax of command lines allows to automatically analyze them, different from analyzing other sources such as operator's free-form documents; (c) Configuration files of current state store correct running status of networks and thus can be correct information used for validation. We aim to validate configuration commands as much as possible within these information sources.

B. Graph-based validation

To balance the trade-off between extensibility and interpretability, we aim to establish a validation methodology based on graph (composed of vertices and edges). Graph is a kind of abstracted object capable of representing referential relationships and more importantly capable of visualization – an essential way to provide interpretability. Since tenant networks can be regarded as virtual resources and connections between pairs of them as described in Sec. II-B, a vertex represents an instance of virtual resource (i.e., VR, VIF, VLAN) and a single edge stands for a referential relationship between virtual resources (as detailed in Sec. IV-A); This graph hence models the network-wide dependency of device configuration settings. In addition, since configuration commands can be regarded as creating virtual resources and connecting a pair of them, we abstract configuration commands as graph-based operations such as addition/deletion of vertices/edges (as detailed in Sec. IV-B).

C. Principle-based validation rules

Another concern is to define the correctness of network configuration. For the generality of configuration validation, we define a series of common rules based on networking

TABLE III. RULES OF CONFIGURATION VIOLATION.

#	Name of violation	Description	Type
v1	double definition/deletion	try to newly define already-existing virtual resources, or try to delete inexisting virtual resources	per-operation
v2	reference to inexisting node	try to connect a virtual resource that has not been defined	per-operation
v3	inter-tenant misconnection	networks of different tenants are connected, resulting in security violation	per-operation
v4	intra-tenant separation	the network of a single tenant is composed of disjoint sub-topologies	network-wide
v5	intra-tenant ID duplication	some network IDs (e.g., IP address) are not unique inside a single tenant	network-wide
v6	intra-tenant reachability	draw a reachability matrix from an interface to another inside a single tenant	network-wide

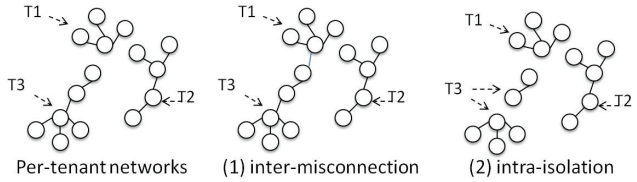

Fig. 3. Examples of principle violation in multi-tenant network context.

principles that any multi-tenant datacenter network must not violate. Table III lists the violation rules and Figure 3 illustrates a few examples of them. We translate these rules into graph-based ones in Sec. IV-D.

The first two rules v1 and v2 are related to the correctness of configuration commands to create/delete a virtual resource. The rule v3 is the most important in the multi-tenant context to assure the separation among tenants. We run validation for these 3 rules in a per-graph-operation manner (as noted in Sec. IV-D). The rule v4 and v5 are related to fundamentals of communication to provide reachability. According to the rule v6, since the correctness of reachability inside a tenant is up to the engineers' intent, we determine to only visualize reachability information (e.g., which pair can communicate and where the communication fails if cannot) so that engineers can identify whether the configuration commands correctly reflect the intent. We examine these 3 validations after executing all the commands as they represent network-wide semantics.

Also, there are protocol-specific validation rules such as BGP routing policy settings in the ISP field. These specific rules have been discussed [6], [7] and such protocol-specific settings are less significant regarding tenant deployment inside datacenters, and thus this paper rather focuses on the above-mentioned common rules as a new primitives of configuration validation for the recently-emerged context (i.e., multi-tenant datacenter network). We envision that those specific validation rules are plugged into the validation procedure in the future.

IV. METHODOLOGY

This section presents the methodology we construct to ease the configuration validation procedure. Overall structure of the methodology is displayed in Figure 4. According to a virtual network model (Sec. IV-A), configuration commands are abstracted as graph operations (Sec IV-B). After examining the existing per-tenant networks identified from exiting configuration files to recover tenant context (Sec. IV-C), new configuration commands are validated in an abstracted graph-based manner (Sec. IV-D). We customize the basic idea of virtual network model (Sec. IV-A) and per-tenant network identification (Sec. IV-C) presented in our previous work [3] (proposed for mining existing network configuration files to find typical patterns – a substantially different purpose); We briefly revisit them in this paper for self-containedness as well.

A. Models of tenant networks

We formally define a tenant topology as an attributed graph $G = (V, E, \Sigma)$, where $V = (v_1, \ldots, v_N)$ is a set of vertices, $E = (e_1, \ldots, e_M)$ is a set of edges (also noted as $e_i = (v_j, v_k)$), and $\Sigma : V \rightarrow L$ is a labeling function that maps a vertex to a label. A vertex in the network can be uniquely differentiated with the label $l \in L$, which is characterized by 3 attributes $l = (t, p, d)$, where t is the type of virtual resource (e.g., VLAN, VR), and p is the parameter of virtual resource (e.g., 10, vrA), d is the ID of device. For example, a virtual router vr3 in core switch CS1 is abstracted as a vertex v_i with the corresponding label $L(v_i) = (t_i, p_i, d_i) = (VR, vr3, CS1)$.

We note that the device id (d) of VLAN vertex should exceptionally be constant across all the devices (e.g., $d = null$), because we regard a VLAN vertex as an aggregated L2 network connecting multiple virtual interfaces across devices. For instance, VIFs of FW and AggrSw displayed in Figure 2 $v_1 = (VIF, FW, eth0/2.20)$ and $v_2 = (VIF, AggrSw, 20)$ are connected via the corresponding VLAN instance $v_3 = (VLAN, null, 20)$ (i.e., $v_1 - v_3 - v_2$). This abstraction of an L2 network as a vertex hides the need for inputting physical topology to obtain current state of topology, which can be conducted when networks are correctly configured; This would be true in the in-service datacenter, where existing deployed tenants are actually and correctly providing services.

B. Abstracting configuration commands as graph operations

Configuration commands for network devices can be basically abstracted as graph operations such as addition/deletion of vertices/edges (Figure 4(a)). Table IV shows several examples of the mapping from a configuration command to a set of graph operations. Here, graph operation is organized with (a) *function* represented as $func\_name(\cdot)$ listed below and (b) *argument vertex* represented as the vertex form $v = (t, p, d)$.

We classify the functions as follows.

- add_node(v) : add the vertex v to the entire network.
- add_edge(v_1, v_2) : add the edge $e = (v_1, v_2)$ to the entire network.
- add_node_if_inexist(v) : add the vertex v to the entire network if the vertex is not previously defined (do nothing if already defined).

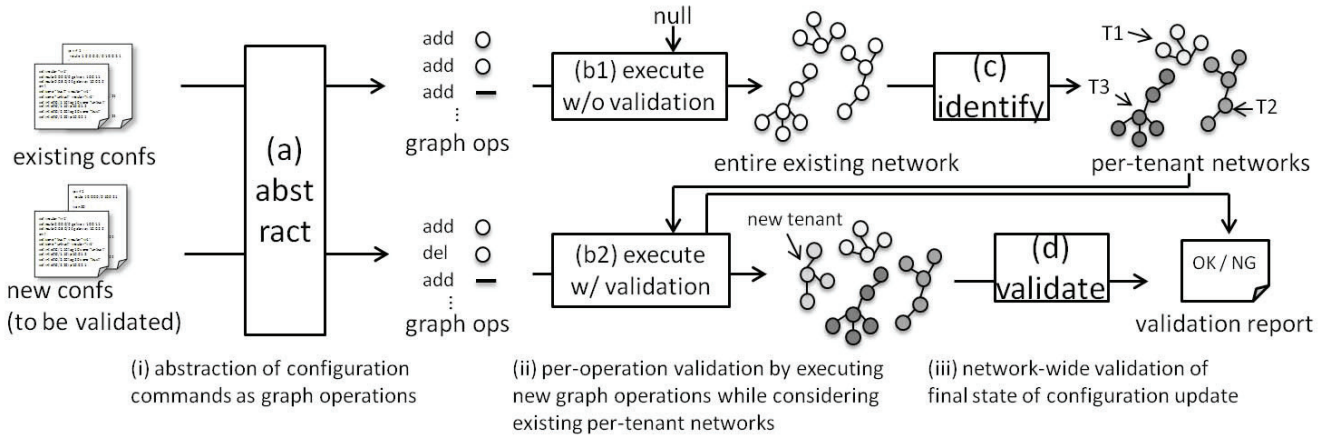

Fig. 4. Overview of the validation procedure.

TABLE IV. ABSTRACTING CONFIGURATION COMMANDS AS GRAPH OPERATIONS.

#	Configuration commands	Graph operation
c1	vlan *vlanId* . . . !	add_node_if_inexist(v) , v = (*VLAN, vlanId, null*)
c2	ip vrf *vrfId* . . . !	add_node_if_inexist(v) , v = (*VR, vrfId, dev*)
c3	interface vlan *vifId* . . . ip vrf forwarding *vrfId* . . . !	delete_edge_if_exist(v_1, v_2) , v_1 = (*VIF, vifId, dev*), v_2 = (*VR, any, dev*) add_node_if_inexist(v) , v = (*VIF, vifId, dev*) add_edge_if_inexist(v_1, v_2) , v_1 = (*VLAN, vifId, null*), v_2 = (*VIF, vifId, dev*)
c4	set vrouter "*vrId*"	add_node_if_inexist(v) , v = (*VR, vrId, dev*)
c5	set zone "*zoneId*" vrouter "*vrId*"	add_node(v) , v = (*ZONE, zoneId, dev*) add_edge(v_1, v_2) , v_1 = (*VR, vrId, dev*), v_2 = (*ZONE, zoneId, dev*)
c6	set interface *vifId* tag *vlanId* zone "*zoneId*"	add_node(v) , n = (*VIF, vifId, dev*) add_node_if_inexist(v) , v = (*VLAN, vlanId, null*) add_edge(v_1, v_2) , v_1 = (*VIF, vifId, dev*), v_2 = (*VLAN, vlanId, null*) add_edge(v_1, v_2) , v_1 = (*VIF, vifId, dev*), v_2 = (*ZONE, zoneId, dev*)

- add_edge_if_inexist(v_1, v_2) : add the edge $e = (v_1, v_2)$ to the entire network if the edge is not previously defined (do nothing if already defined).
- delete_edge_if_exist(v_1, v_2) : delete the edge $e = (v_1, v_2)$ from the entire network if any.

Those functions reflect the variety of configuration manners for different types of (often vendor-specific) network devices. For example, the command c6 "set interface . . ." for a certain type of device only represents the allocation of a VIF instance so that the secondary execution of the same command will be rejected by the device; This behavior is characterized by the add_node(v) operation. On the other hand, the command c2 "ip vrf *vrfId* . . ." for another type of device means to create a VR instance as well as to change configuration mode to detailed VR setting, and thus secondary declaration of this command does not care the existence of the VR instance, leading to the add_node_if_inexist(v) operation. Also, the command c3 "interface vlan *vifId* . . ." overwrites the existing reference between VIF and VR if any, resulting in the requirement of the delete_edge_if_exist(v_1, v_2) operation.

The argument vertex $v = (t, p, d)$ is composed of the 3 attributes (Sec. IV-A), which are identified as follows. The type of virtual resource t is identified based on the command syntaxes, the parameter of virtual resource p is extracted from specific areas in commands, and the ID of device d corresponds to the name of configured device. We set additional property based on the configuration commands if necessary. For example, if we find an "ip address x.x.x.x y.y.y.y" command for a VIF instance (e.g., line 20 in Figure 2), then the IP address and netmask parameters are set on the VIF vertex. Also, for another example, if we find an "ip route vrf *vrfId* x.x.x.x y.y.y.y z.z.z.z" command for a VR instance (e.g., line 12 in Figure 2), then this routing entry is set in the VR instance. These parameters are used to identify the duplicated IP addresses and to compute IP reachability (Sec. IV-D).

Similarly, such graph operation could represent deletion commands such as "no interface vlan *id*" to delete a VIF instance, or "unset vrouter *vrId*" to remove a VR instance. Although deletion is also an important update, the abstraction is not displayed in this paper due to the lack of space.

C. Identifying existing tenant networks

Before executing the validation procedure, we need to identify per-tenant networks existing in the physical infrastructure, which is required to examine inter- or intra-tenant validation

TABLE V. PER-OPERATION VALIDATION RULES TRANSLATED AS GRAPH OPERATION CONTEXT.

#	Operation	Validation rules
r1	add_node(v)	if vertex v already exists in the current state ($v \in V_e$), then error (v1: double definition)
r2	add_edge(v_1, v_2)	if edge e = (v_1, v_2) already exists ($e \in E_e$), then error (v1: double definition) if vertex v_1 or n_2 does not exist ($v \notin V_e$), then error (v2: referential error) if v_1 or v_2 belong to different tenants ($T(v_1) \neq T(v_2)$), then error (v3: inter-tenant isolation)
r3	add_node_if_inexist(v)	n/a
r4	add_edge_if_inexist(v_1, v_2)	if vertex v_1 or v_2 does not exist ($v \notin V_e$), then error (v2: referential error) if n_1 or n_2 belong to different tenants ($T(v_1) \neq T(v_2)$), then error (v3: inter-tenant isolation)

context. Per-tenant networks are not obvious as there is no semantics of tenants in the low-level device configuration commands. In addition, there is no direct mapping between tenant networks and configuration files, because a tenant network is composed of virtual resources of several devices whereas a device allocates virtual resources for multiple tenants.

We obtain per-tenant networks as follows: (1) Executing the graph operations (addition of vertices/edges) extracted from current configuration files to reproduce the entire network (Figure 4(b1)), which we note as $G_e = (V_e, E_e, \Sigma)$; (2) Using a conventional graph traversal method over the entire network G_e to capture connected components, each of which corresponds to a tenant network (Figure 4(c)). This second step is plausible because any pair of tenant networks must not be inter-connected due to the principle of multi-tenant networking. Details can be found in Ref. [3].

We obtain a set of graphs representing existing per-tenant networks $(G_1, G_2, \ldots, G_N)$, where N is the number of existing tenants ($G_i = (V_i, E_i, \Sigma)$). All the vertices inside a tenant network G_i is colored by the ID of that tenant, i.e., for all $T(v) = i$ for $\forall v \in V_i$ of G_i with a function $T : V \rightarrow N$.

D. Validating tenant-level configurations

1) Per-operation validation: We first validate the new configuration commands in the per-operation manner (Figure 4(b2)). Here, we define per-operation validation as the one which should be examined at the moment when a command is executed (not at the moment of final state after executing all the commands). The primary motivation for per-operation validation is to ensure the correctness of configurations for any snapshot during updates (not only final state). For example, let us consider an update including several addition/deletion of virtual resources inside a single tenant. Such update would cause errors such as inter-tenant misconnection, if these operations were executed in a certain wrong order even though the resulting final snapshot suggests the correct status.

Table V lists the validation rules (defined in Sec. IV-B) for each graph operation. The execution of a single operation corresponds to a set of validation rules. For example, add_edge(v_1, v_2) is related to a few validation rules including double definition (v1), reference to inexisting node (v2), inter-tenant misconnection (v3), each of which can be formally defined with graph-based conditional expression as shown in the table. After passing these validation rules, we execute the command to update the network to the next state.

2) Network-wide validation: Network-wide validation can be performed as follows (Figure 4(d)).

The intra-tenant separation (v4) means that the per-tenant topology consists of multiple disconnected components, which is an unusual case to provide services. The detection of this violation is achieved by checking whether $G_i = (V_i, E_i, \Sigma)$ is a single connected component or not. For example, one can select an arbitrary vertex $v \in V_i$ and then use a conventional graph traversal method to obtain the connected component including v; If this connected component is nonequivalent to G_i then this graph is regarded as violating the intra-tenant separation rule.

The duplication of ID (v5) can be easily detected. We select a VLAN node, which connects to a set of VIF instances belonging to the corresponding IP segment. If there is an identical IP address assigned for different VIFs, then this graph is considered as violating the ID duplication rule.

The reachability matrix (v6) can be drawn as follows. We select a pair of source/destination VIFs, and compute the reachability of a packet having source/destination IP addresses of the VIFs by simulating routing procedure inside the tenant network. We check whether the packet starting at the source VIF can reach the destination VIF. Examination of the reachability of any pair of two VIFs result in the reachability matrix, which is eventually visualized as a table format.

V. EVALUATION

A. Dataset

We create evaluation datasets synthesized from configuration files of an actual datacenter. This datacenter provides a multi-tenancy hosting service for business uses, leading to a number of deployed tenants. The virtual networks for tenants are variously composed of diverse sets of virtual resources, resulting in more than 10,000 lines of configuration commands in total.

A set of current configuration files, which is an input of the method, is reproduced from the original configuration files of switches and firewall. We reproduce several datasets of different numbers of deployed tenants to evaluate the scalability of the method against the number of tenants as follows. We identify the N per-tenant networks (i.e., N graphs $G_1, \ldots, G_N$) inside the network configurations following the method described in Sec IV-C, and regenerate several sets of configuration files containing only M tenants ($M < N$) randomly selected from all the identified ones (M out of those N graphs are re-converted into the corresponding synthesized configuration commands).

For the set of new configuration commands, which is the other input of the method, we select a typical pattern of tenant construction found in Ref. [3], consisting of configuration commands of creating a number of VLANs, VIFS, and VRs for both switches and firewalls, leading to over 50 lines.

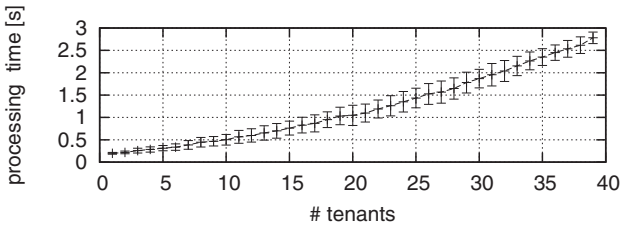

Fig. 5. Performance evaluation.

TABLE VI. TIME TO VALIDATE CONFIGURATION FILES FOR THREE TESTEES.

	testee A	testee B	testee C
w/o tool	7.17 min.	13.17 min.	13.92 min.
w/ tool	4.33 min.	3.88 min.	1.88 min.

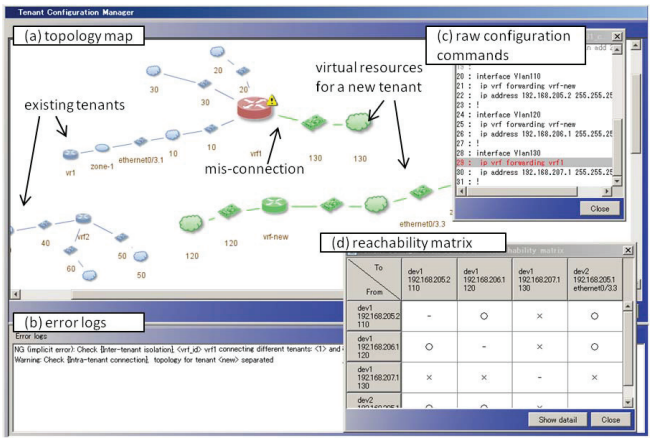

Fig. 6. Screenshot of a prototype implementation.

B. Computational practicability

Figure 5 displays the processing time of the validation methodology, measured with a 2.1GHz CPU. This figure shows the average and standard deviation with the 100 trials of running a prototype implementation for each M. A trail is conducted with existing configuration files of randomly-selected M tenants from the original data.

This figure shows that the methodology is lightweight so that it can be repetitively used with a personal computer for an engineer. The main reason for the light computational cost is that most of validation rules can be examined within per graph operation (i.e., rules v1–v3) or per-single tenant network (i.e., rules v4–v6), avoiding the need for dealing with the entire network including all the M tenants (forming a vast search space); Indeed, we found that most of the computational time were derived from the identification of exiting per-tenant networks (Sec. IV-C).

C. Operational practicability

Workload mitigation. We measured the workload of configuration validation performed by 3 testees, who have knowledge about configuration syntax and construction of tenant virtual network. We randomly injected parameter errors into the above-mentioned new configuration commands [3]; The number of errors was between 0 and 3 as we consider that actual network engineers will not simultaneously yield a high number of errors, and was not informed to the testees (which reflects actual cases). The testees had to detect all the errors and to make sure that there was no remaining errors. Before the measurement, we preliminarily reviewed the meanings of commands and structure of tenants, and ran a trial of the validation procedure to acquire how to use the validation tool, which took about 1 hour for each testee. The results for the actual validation tasks are displayed in Table VI; The testees took 11.42 ± 3.02 minutes without the tool, whereas they took 3.63 ± 1.10 minutes with the tool on the basis of average and standard deviation (approximately 68% reduction). The testees found that the visualization of configuration commands (shown later) allowed them to easily and promptly identify the existence of errors and to detect the corresponding command.

Interpretability. Figure 6 exhibits a screenshot of a prototype implementation of standalone graphical program with the input of incorrect configuration commands; This program demonstrates several capabilities for providing interpretation. For instance, in case of inter-tenant misconnection, corresponding nodes are highlighted to help to detect which virtual resources of which devices are actually about to cause this violation, as shown in Figure 6(a). Also, this figure shows that virtual resources for a new tenant are separated into two groups, meaning the incorrect separation inside a tenant network. This program also displays a few other independent windows showing raw configuration commands examined for validation. In these windows, wrong commands are colored, as shown in Figure 6(c), and the mouse-clicks over those wrong command lines highlight the corresponding virtual resources in the graph figure. This tool also displays reachability between every pair of VIFs as shown in Figure 6(d); This window can further exhibit at which VR the communication between a pair fails if not reachable (any why), which helped the testees to pinpoint parameter errors in routing setting.

Extensibility. We additionally investigated the configuration commands of a load balancer and a VPN device, each of which is provided by a company different from the ones for the switch and firewall used throughout this paper. We found that both the two additional types of devices have the capability of VLAN, VIF, and VR, and that corresponding configuration commands can be interpreted as creation/connection of those virtual resources. Hence, to extend the proposed methodology to adopt those new devices, we only have to relate configuration commands to graph operations. On the other hand, writing a number of device-specific validation rules for those devices without such abstraction would lead to the difficulty in writing and maintaining correct rules.

[3]Specifically, for the configuration files of switches and firewalls, (a) we identify parameters such as VLAN ID, VIF ID, Zone ID, VRF/VR ID, IP addresses of VIFs, and IP addresses and network addresses of routing entries by parsing those configuration files, (b) the error injector we implemented selects pre-defined number of parameters out of the identified ones in the configuration files and intentionally replaces the value of the selected parameters with other values while preserving their possible ranges (e.g,. VLAN ID should be between 1 and 4095), which would represent errors of parameter assignment.

VI. Discussion

In order to achieve practical extensibility, we designed this methodology as extensible in a pluggable manner accepting per-device modules. A module includes functionalities of parsing a configuration file, identifying (or parsing) specific commands representing addition/deletion of virtual resources, and abstracting these commands as graph operations. This abstraction requires the module developer to inspect the behavior of devices regarding the input of those commands. For instance regarding the behavior, a command of adding a connection between resources might be rejected if one of the resources is already connected to yet another one (resulting in the add_edge_if_inexist(v_1, v_2) operation), whereas another command might be accepted while overwriting the existing connection (resulting in the additional use of the delete_edge_if_exist(v_1, v_2) operation). Still, as a lesson learned in this work, we faced some cases that the investigation of device behavior was not necessarily obvious (e.g., a command overwrites some existing settings but rejects to overwrite other existing ones). Also, it is generally cumbersome to parse such configuration commands in a general way as claimed in Ref. [8], although device commands are intuitive and well-designed. We envision that device vendors would provide API-like specification including their devices' behaviors against the instructions of setting changes (both for ordinal and exceptional cases), which we believe should be a key characteristics of future network management paradigm.

VII. Related work

Statistics-based approach. A few works have proposed methods of detecting configuration anomalies based on probabilistic approaches such as Bayesian statistics [9] and association mining [10], defining anomalies as statistical outliers. Although such statistical approaches can possibly uncover unknown anomalies in essence, their undeterministic nature cannot necessarily assure that the examined configurations satisfy specific explicit conditions. As a result, operators still have to inspect their configuration for the completeness of validation. Instead, we focus on deterministic approaches as a complementary and prior one for configuration validation, envisioning to develop a relevant framework leveraging the advantages of both the two approaches.

Configuration validation of specific protocols or devices. A basic approach to deterministically validate configurations is to enumerate detailed rules that those configurations must follow [11], [6], [7], [12], [13]. These works define a number of rules for VPN and for BGP routing in wide-area networks, with which operators can effectively interpret which rules are violated by which commands. Other works deal with detailed aspects of specific devices such as firewalls [14], [15], [16]. Complementary to those works, we rather focus on extensibility of validation, which is one of the key features for practical use, because datacenters consist of various devices. We envision to integrate those specific rules with this work in the future work (e.g., by using validation-oriented language such as Refs. [12], [16]).

General abstraction for validation. A few works have established original algebraic frameworks designed for general applicability to various networking contexts. Ref. [4] reduces a validation problem to a Boolean satisfiability problem (SAT), examining the entries of Forwarding Information Base (FIB). Ref. [5] defines yet another original Boolean algebraic system to represent networking primitives. Also, Ref [17] develops a model-checking framework based on Binary Decision Diagram (BDD) and Computation Tree Logic (CTL) to test network properties. The primary shortcoming from these high-level abstraction, however, is the lack of interpretability, whereas our approach is designed to preserve interpretability by using visualizable graphs as effective as possible.

Pre-deployment validation. Overall, those past works mentioned thus far mostly deal with configuration files obtained with a certain snapshot of device state – not with including new configuration commands that have to be validated before being input into devices. Indeed, a few of those works have mentioned the possibility of such pre-deployment validation, but unfortunately there were not detailed statement about how to realize this such as how to model network updates considering the actual vendor-specific behaviors of network devices regarding the execution of configuration commands; Indeed, abstracting the execution of actual device commands is not a simple or easy problem as described in this paper.

Validation of emerging technology. Refs. [18], [19], [20] have constructed effective frameworks for validating the recently emerging networking primitives, i.e., OpenFlow. Different from these works, which are essential for the future environments, we rather focus on today's existing ones.

VIII. Concluding remarks

In this work, we presented a methodology to statically validate network device configurations to deploy a tenant without actually deploying it. The basic approach was to predict a posteriori network states (i.e., states after execution of each configuration command), which are applied to validation rules. We mainly made use of a graph-based approach to provide both extensibility to deal with various types of network devices consisting of datacenters (provided from abstraction of configurations as graphs), as well as interpretability to easily understand the validation results (provided from visualization – a key benefit of using graphs). The evaluation with a dataset synthesized from a set of actual configuration files resulted in 68% reduction of the time to validate them on average. A future work should be to further evaluate the effectiveness of our approach by comparing it with others.

Acknowledgement

We especially appreciate Takashi Tashiro of Hitachi Systems, Ltd., Hitoshi Nishikawa of Hitachi Solutions, Ltd., Kazuma Yumoto, Masaya Umemura, Mariko Yamada, Yoji Ozawa of Hitachi Yokohama Research Laboratory, and all the anonymous reviewers of IM for invaluable comments and suggestions.

REFERENCES

[1] "Cisco Systems Inc., Cisco Virtualized Multi-Tenant Data Center, Version 2.1 Implementation Guide." [Online]. Available: http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/VMDC/2.1/implementation_guide/vmdcImpl21.pdf

[2] "Juniper Networks Inc., Cloud Ready Data Center Network Design Guide." [Online]. Available: http://www.juniper.net/us/en/local/pdf/design-guides/8020014-en.pdf

[3] Y. Himura and Y. Yasuda, "Discovering Configuration Templates of Virtualized Tenant Networks in Multi-tenancy Datacenters via Graph-mining," *ACM SIGCOMM CCR*, vol. 42, no. 3, pp. 13–20, July 2012.

[4] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the Data Plane with Anteater," in *ACM SIGCOMM'11*, August 2011, pp. 290–301.

[5] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking For Networks," in *USENIX NSDI'12*, April 2012, p. 14.

[6] N. Feamster, "Practical Verification Techniques for Wide-Area Routing," *ACM SIGCOMM CCR*, vol. 34, no. 1, pp. 87–92, January 2004.

[7] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *USENIX NSDI'05*, May 2005, pp. 43–56.

[8] D. Caldwell, S. Lee, and Y. Mandelbaum, "Adaptive Parsing of Router Configuration Languages," in *IEEE INM'08*, October 2008, pp. 1–6.

[9] K. El-Arini and K. Killourhy, "Bayesian Detection of Router Configuration Anomalies," in *ACM SIGCOMM MineNet'05*, August 2005, pp. 221–222.

[10] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Minerals: Using Data Mining to Detect Router Misconfigurations," in *ACM SIGCOMM MineNet'06*, September 2006, pp. 293–298.

[11] R. Deca, O. Cherkaoui, and D. Puche, "A Validation Solution for Network Configuration," in *IEEE CNSR'04*, May 2004, p. 3.

[12] L. Vanbever, G. Pardoen, and O. Bonaventure, "Towards validated network configurations with NCGuard," in *IEEE INM'08*, October 2008, p. 6.

[13] S. Lee, T. Wong, and H. S. Kim, "Netpiler: Detection of Ineffective Router Configurations," *IEEE JSAC*, vol. 27, pp. 291–301, April 2009.

[14] E. Al-Shaer and H. H. Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," in *IEEE INFOCOM 2004*, March 2004, pp. 2605–2616.

[15] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "FIREMAN: A Toolkit for FIREwall Modeling and ANalysis," in *IEEE S&P 2006*, May 2006, pp. 199–213.

[16] J. Lobo and V. Pappas, "C^2: The Case for a Network Configuration Checking Language," in *IEEE POLICY'08*, June 2008, pp. 29–36.

[17] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi, "Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security," in *IEEE ICNP'09*, November 2009, pp. 123–132.

[18] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures," in *ACM SafeConfig'10*, October 2010, pp. 37–44.

[19] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," in *ACM SIGCOMM'12*, August 2012, p. 12.

[20] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test OpenFlow Applications," in *USENIX NSDI'12*, April 2012, p. 14.