Software-Defined Network Assimilation: Bridging the Last Mile Towards Centralized Network Configuration Management with NAssim

Huangxun Chen[1], Yukai Miao[2], Li Chen[3], Haifeng Sun[4], Hong Xu[5], Libin Liu[6], Gong Zhang[1], Wei Wang[7,8]
[1]Huawei Theory Lab, [2]University of New South Wales, [3]Zhongguancun Laboratory,
[4]Beijing University of Posts and Telecommunications, [5]Chinese University of Hong Kong,
[6]Shandong Computer Science Center (National Supercomputer Center in Jinan),
[7]Hong Kong University of Science and Technology (Guangzhou), [8]Hong Kong University of Science and Technology
chen.huangxun@huawei.com,yukai.miao@unsw.edu.au,lichen@zgclab.edu.cn,hfsun@bupt.edu.cn
hongxu@cuhk.edu.hk,liu.libin@outlook.com,nicholas.zhang@huawei.com,weiwcs@ust.hk

ABSTRACT

On-boarding new devices into an existing SDN network is a pain for network operations (NetOps) teams, because much expert effort is required to bridge the gap between the configuration models of the new devices and the unified data model in the SDN controller. In this work, we present an assistant framework NAssim, to help NetOps accelerate the process of assimilating a new device into a SDN network. Our solution features a unified parser framework to parse diverse device user manuals into preliminary configuration models, a rigorous validator that confirm the correctness of the models via formal syntax analysis, model hierarchy validation and empirical data validation, and a deep-learning-based mapping algorithm that uses state-of-the-art neural language processing techniques to produce human-comprehensible recommended mapping between the validated configuration model and the one in the SDN controller. In all, NAssim liberates the NetOps from most tedious tasks by learning directly from devices' manuals to produce data models which are comprehensible by both the SDN controller and human experts. Our evaluation shows, NAssim can accelerate the assimilation process by 9.1x. In this process, we also identify and correct 243 errors in four mainstream vendors' device manuals, and release a validated and expert-curated dataset of parsed manual corpus for future research.

CCS CONCEPTS

• **Networks** → **Network manageability**; **Network management**; • **Computing methodologies** → *Machine learning*;

KEYWORDS

Multi-Vendor Networks, Network Configuration Management, Software-Defined Networks

ACM Reference Format:
Huangxun Chen[1], Yukai Miao[2], Li Chen[3], Haifeng Sun[4], Hong Xu[5], Libin Liu[6], Gong Zhang[1], Wei Wang[7,8]. 2022. Software-Defined Network Assimilation: Bridging the Last Mile Towards Centralized Network Configuration Management with NAssim. In *SIGCOMM '22: SIGCOMM 2022, August 22-26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3544216.3544244

1 INTRODUCTION

Software-defined networking (SDN) is the prevalent approach to manage modern enterprise and cloud networks [39, 52]. The main characteristic of the SDN is centralized control over all network devices, physical and virtual. This is easy to achieve for a newly constructed network with devices that can run software agents [2, 8] accessible to a SDN controller. However, for a network with many legacy devices, current SDN controllers must use their command line interfaces (CLI) to gain access. Enabling CLI-based configuration for SDN controllers requires significant effort from network operations (NetOps) experts, who have to understand the device's user manuals, find correct commands, draft and validate configuration templates, and provide rules that map parameters from the SDN's configuration database to the templates. Apart from legacy devices, on-boarding new vendors and introducing new devices from them is also painstaking. Due to the lack of standardized interface for configuration and operation, NetOps teams need to make significant effort in adapting new devices to existing SDN controllers by either developing software agents, or, if the device has only CLI access, going through the same process above.

We define the process of introducing heterogeneous network devices (e.g. legacy devices and devices from a new vendor) into a centrally controlled, existing SDN network as *Software-defined Network Assimilation* (SNA). We find that current SNA approaches requires significant human effort and are error-prone. Thus, we seek to take the SDN through the last mile towards centralized configuration management across all devices, legacy and new.

The key problem in SNA is the mismatch between two data models: the heterogeneous configuration model of the device, and

The work was done when Yukai Miao was an intern at Huawei.
Corresponding author: Li Chen and Haifeng Sun.

SIGCOMM '22, August 22-26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544244

the unified device model (UDM) [16, 49] in the centralized SDN controller. Our insight to tackle this problem is to learn from the current SNA practices of NetOps: to assimilate a new device, NetOps engineers first prepare command templates by reading and understanding the user manual of the device, and then link the parameters in the command templates to their counterparts in the UDM. To accelerate the current practice, our proposal, NAssim, also has two phases: the Vendor-specific Device Model (VDM) construction phase, and the VDM-UDM mapping phase. In the first phase, NAssim helps NetOps engineers to extract and validate the VDM from its manual; and in the second, NAssim helps to map the parameters in the VDM to UDM.

This involves solving three challenges:

- **Manual Format Heterogeneity:** Devices' user manuals are the most reliable sources to extract their configuration models, especially for legacy devices which have CLI only. However, the manuals are organized and formatted uniquely for different vendors. Thus, we need a unified parsing framework to extract configuration models from manuals.
- **Errors & Ambiguity in Manuals:** Despite being the single source of truth, the manuals contain many errors and ambiguities, thus only parsing from manuals is not enough. To validate and correct parsed models, we must seek an efficient way to identify errors and clarify the ambiguities.
- **Heterogeneity between Configuration Models:** Configuration data models are different for different vendors. To map them to an UDM is challenging, and, we believe, must involve using natural language processing (NLP) techniques to learn from the natural language descriptions of the CLI commands in user manuals.

We build NAssim as an assistant framework for NetOps engineers to address the above challenges. NAssim consists of three core components: Parser Framework, Validator, and Mapper. The workflow is as follows: to assimilate a new device, first NetOps engineers specify a parser for its user manual using the Parser Framework, and run the parser to extract a preliminary VDM from the manual; then, we use the Validator to confirm the correctness of the model via formal analysis, model hierarchy validation, and empirical data validation; any errors and ambiguity in the manual are caught in the Validator, and are summarized and reported to engineers for corrections; finally, between the VDM and the SDN controller's UDM, the Mapper uses a novel NLP algorithm to produce a recommended mapping which is human-comprehensible. For each relationship in the mapping, if NetOps engineers find it questionable, NAssim can also present a list of most relevant recommendations with rich semantic information parsed from the manual, so that they no longer need to search through the manuals themselves.

We enable this workflow with the following contributions:

(1) We build a parser framework to enable customized parsing of user manuals of network devices. With a comprehensive study of commonalities and diversity of popular vendors' manuals, we design a vendor-independent device model corpus format which is expressive, interpretable, and extendable. Using this format, we enable the Test-Driven Development of parsers, which enhances the reliability of device manual parsing. In our experience of parsing the manuals from mainstream vendors, it requires ~50 lines of codes per vendor.
(2) Since the parsed results still contain typos or other human errors in the manuals, we design a rigorous and human-comprehensible validation framework for the parsed results, which has three stages:
    (a) *Formal syntax validation:* We transform all parsed command styling conventions into Backus Normal Form and build corresponding syntax parsers to verify the conformance of all parsed commands.
    (b) *Model hierarchy validation:* To validate hierarchy among commands and identify missing ones, we construct a graph model for each CLI instance, and design a state-machine-based template matching algorithm for validation.
    (c) *Validation with empirical data:* Finally, as an end-to-end testing scheme, we use configurations from running devices to check the correctness of the parsed results. We also present a scheme to generate abundant empirical data, using the aforementioned command graph model to test against real devices.

    In our evaluation with 613 real configuration files, we achieve 100% matching accuracy between configuration snippets and the validated model. NAssim's Validator also identifies 184 syntactic errors and 59 ambiguities in four mainstream vendors' manuals.
(3) We release a parsed, validated, and expert-curated dataset of device manual corpus of different vendors[1] for future research in the SDN and network management.
(4) We employ state-of-the-art contextual learning NLP model, BERT, and augment it to create a domain-adapted version for our usecase, NetBERT. We use NetBERT to map any parsed configuration model to the UDM. NetBERT achieves 89% and 70% top 10 recall for mapping device models of Huawei and Nokia to a given UDM respectively. The output of NetBERT is a mapping between parameters of different models, which is comprehensible and editable by NetOps experts.

In what follows, we overview the background in § 2, and present the overall design of NAssim in § 3. We then describe its three components in details in § 4, § 5, &§ 6. We evaluate NAssim in § 7, and discuss related work in § 9.

Caveat: To the best of our knowledge, this is the first study of network configuration manuals, despite its importance in network management and operations. We find that the manuals, as human-written documents, contain non-negligible errors which, if followed blindly, can not only impede the SNA process, but also lead to production issues. Correcting these errors is not trivial, and need judgement and trial-and-error experimentation by human experts. This is also true for the VDM-UDM mapping task. Therefore, we believe SNA cannot be easily automated, and we design NAssim as an assistant SNA framework for NetOps engineers. NAssim provides tools and recommendations to NetOps engineers to accelerate the current SNA process by automating the most tedious tasks, and is not an end-to-end system that directly extract and map heterogeneous VDM to UDM.

[1] https://github.com/AmyWorkspace/nassim

Disclaimer: This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

In this section, we first highlight the need for SNA in modern SDN networks (§2.1). Next, we summarize the main challenges for SNA (§2.2). Finally, we motivate and explain the design decisions of NAssim (§2.3).

2.1 SDN & Need for SNA

Large enterprises procure and operate network devices from multiple vendors. They also continuously introduce new device models and new vendors into their network. This multi-vendor nature makes the network management tasks complex [7, 9, 20, 22, 33, 33, 34, 40, 42, 44]. To ease the difficulty of managing a multi-vendor network, the NetOps community has been relying on SDN, which presents a logically centralized control plane for all network devices. At the core of SDN, the key data structure is a unified device model (UDM) which normalizes the configuration model of devices from all vendors. Combined with the relevant tools [2, 8], the SDN controllers can configure multi-vendor devices as if they are the same. NetOps of large enterprises and cloud providers have put much effort into building UDMs[10, 16, 49].

While UDM makes the management of existing devices easy, it also makes on-boarding new vendors and introducing devices with differing configuration models difficult. Since each SDN network may have a different UDM, vendors usually do not have a standardized north-bound interface for configuration and operations, despite continuous efforts in this area [8, 10, 16, 49]. This situation is worse for legacy devices with only CLIs. For these devices, SDN controller need to send CLI commands via Telnet connections to implement high-level operational intents. Constructing a valid CLI commands also requires the SDN controller to "understand" the CLI syntax, as well as having a mapping between the UDM and the device's own configuration model.

To the best of our knowledge, currently SNA rely solely on NetOps engineers, who handcraft models for new and legacy devices based on their user manuals, and map the model to UDM. The entire process requires significant human effort and is error-prone. Thus, we seek a system that can accelerate SNA by learning the device configuration model from the manuals with minimal expert effort, and can map the learned model to the UDM in a human-comprehensible manner.

2.2 Challenges of SNA

To this end, we outline three key challenges of SNA.

Manual Format Heterogeneity: Devices' user manuals is usually the most trustworthy sources to obtain configuration models, especially for legacy devices with only CLI access. There is no standardization for manuals, and vendors organize their user manuals in diverse ways, as shown in the manuals of mainstream vendors [4, 11, 13, 15] and the page snapshots in Appendix A. Despite diverse styles, they serve the same purpose: show how to configure the devices via CLI. Thus, the user manuals should cover the CLI commands supported by the devices and the commands' function descriptions, parent/working views, parameter descriptions and representative examples. However, the same concept may have different names in each manual *e.g.*, all manuals specify the parent views of CLI commands (*i.e.*, the working view under which the commands can be accepted and executed), but may put them under 'Views', 'Command Modes', 'Context' and 'View' sections respectively. We survey and summarize five major attributes of most command reference manuals, and their corresponding Cascading Style Sheet (CSS) class names in the online version of manuals for four mainstream vendors in Table 1.

Using the CSS class names seems a trivial way to extract information from online manuals, due to the limited number of mainstream vendors. On the contrary, our study shows that, even for the same attribute, its CSS class name within the same vendor can be inconsistent. Take the Cisco's online manual as an example. In most pages, it stylizes the CLI commands with `'pCE_CmdEnv'` class, while some pages use `'pCENB_CmdEnv_NoBold'`. To distinguish place-holder parameters and keywords in CLIs, different pages in one manual may use one class from candidates including `'cKeyword'`, `'cBold'` and `'cCN_CmdName'`. We suspect, because of the sheer volume of manual pages (10k+ CLIs per manual) for a single device, the manual is written over a long period of time, and the styling and formatting guidelines change over time. The class names are crucial for parsing completeness, otherwise the parsed corpus miss important information in the manuals. However, it is time-consuming, if not impossible, to collect all variants.

Errors & Ambiguity in Manuals: We find that manuals contain errors and ambiguities, and blindly following the manual can impede SNA and even cause production issues. For example, we find an ambiguous CLI command template in a Cisco manual[2]:

```
neighbor { <ip-addr> | <ip-prefix/length> }
   [ remote-as { <as-num> [ <.as-num> ] |
   route-map <name> }
```

For the unpaired left bracket before the `remote-as` symbol, there are multiple potential valid options: removing the left bracket, adding a right bracket after `remote-as` symbol, adding a right bracket at the end of the CLI; choosing which requires judgement from experts. However, these problems are hard to catch by human eyes, thus it is impractical for NetOps teams to audit the manuals from first page to the last to identify all problems.

Heterogeneity between Configuration Models: Configuration models are different for each vendor, and this is intentional: the configuration language is designed by each vendor to be visibly different than the competitors', and the syntax is also heavily protected by patents [5].

For the same concept or intent, different vendors use different wordings and syntaxes. Table 2 shows the comparison for the VLAN and spanning tree commands for Cisco, Huawei, and Juniper. Each line in the table shows the command for the same intent in each vendor. We can see that even for simple tasks the configuration commands are significantly different. Thus, matching these diverse configuration models to UDM is challenging.

2.3 Motivating NAssim's Design Decisions

We design NAssim to tackle the three challenges of SNA. In this section we overview NAssim's design decisions.

[2]https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus5500/sw/command/reference/unicast/n5500-ucast-cr/n5500-bgp_cmds_n.html

Attribute \ Vendor	Huawei	Cisco	Nokia	H3C
CLIs	<class="sectiontitle">Format	<class="pCE_CmdEnv">	<class="SyntaxHeader">Syntax	<class="Command">Syntax
FuncDef	<class="sectiontitle">Function	<class="pB1_Body1>	<class="DescriptionHeader">Description	<class="Command">Description
ParentViews	<class="sectiontitle">Views	<class=pCRCM_CmdRefCmdModes> Command Modes	<class="ContextHeader">Context	<class="Command">View
ParaDef	<class="sectiontitle">Parameters	<class="pCRSD_CmdRefSynDesc"> Syntax Description	<class="ParametersHeader">Parameters	<class="Command">Parameters
Examples	<class="sectiontitle">Examples	<class="pCRE_CmdRefExample"> Examples	/	<class="Command">Examples

Table 1: Diversity of Device User Manuals: The 'CLIs' field denotes the formal syntax of CLI commands, which are command templates with place-holder parameters and special characters to specify selection or optional branches. The 'ParaDef' field contains the implication and value range of place-holder parameters. The 'FuncDef' field describes the functionality of the complete CLI. The 'ParentViews' field indicates the parent/working views of CLIs, *i.e.*, one CLI may have multiple viable working views. The 'Examples' field shows examples of common snippets, *e.g.*, entering a parent view and issuing an instantiated CLI.

Intent	Cisco	Huawei	Juniper
check vlan	show vlan [vlanid]	display vlan [vlanid]	show vlan-id/vlans [vlanid]/[vlanname]
add/delete vlan	vlan [vlanid]/no vlan [vlanid]	vlan branch [vlanid]/undo vlan branch [vlanid]	set vlan-id [vlanid]/delete vlan-id [vlanid]
configure spanning tree root bridge	spanning tree vlan [vlanid] root primary	stp instance [vlanid] root primary	spanning-tree vlan-id [vlanid] root primary

Table 2: Configuration syntax comparisons across Cisco, Huawei, and Juniper

To handle heterogeneity in multi-vendor manuals, we survey and study the online version of the manuals, and design a vendor-independent corpus format, which can consolidate diverse manual styles. Since the number of vendors is few (usually less than 10 for a typical Invitation for Bidding of network infrastructure) and each vendor's manual formatting is roughly the same for all their devices, we believe the human effort in building a vendor-specific parser is acceptable. Therefore NAssim's Parser Framework focuses on the development process, and adopts Test-Driven Development (TDD) methodology. For each vendor, the Parser Framework first generates tests based on the type restrictions in the vendor-independent corpus format. The NetOps teams can then compose basic parsing components and configure CSS class names to build a customized parser. NAssim's TDD utilities generates test reports for each version of the parser, which guides the developer to improve the parser. In our experience of parsing mainstream vendors, this human-in-the-loop workflow both accelerates the SNA process and improves the quality of the parsed corpus.

To tackle the second challenge of errors and ambiguity in parsed corpus, we perform validation on three levels: command-level, inter-command-level, and snippet-level. For command-level correctness, we adopt formal syntax validation techniques, and verify whether the parsed command follows the syntax of the manual. For inter-command correctness, we build graph models for commands, and design a state-machine-based matching algorithm to discover and validate hierarchical relationship between commands. For snippet-level correctness, our innovation is to include configuration files/snippets from running devices to validate extracted models, because the only way to ensure operational correctness is to run the exact commands on the actual device. We use two data source for validation: 1) collected configuration files/snippets from running devices; 2) for commands not used by any running devices, we generate configuration snippets from the graph model of commands, and test them on real devices via CLI.

For the last challenge, to map extracted model to UDM, NAssim's innovation is two-fold: 1) we choose to perform fine-grained parameter-level mapping, instead of command-level or snippet-level mapping; doing this also results in a human-comprehensible output of parameter-to-parameter mapping between VDM and UDM, which allows NetOps experts to directly understand and modify; 2) we employ recent NLP advances to generate context encoding of each parameter using all relevant semantic information, and map them using a similarity measure. The similarity-based approach can save valuable time for NetOps experts when they find errors in the mapping, because NAssim can provide a list of most relevant parameters along with their rich semantic information parsed from manuals, so that experts no longer need to search through the entire user manual for such information.

3 NASSIM OVERVIEW

NAssim helps NetOps engineers in two main phases of SNA: VDM construction phase and VDM-UDM mapping phase. In these phases, we use NAssim's three core components as shown in Figure 1.

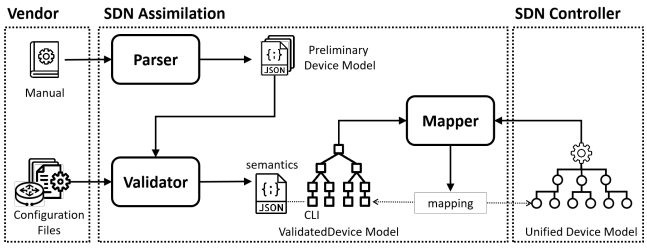


Figure 1: NAssim Overview.

3.1 VDM Construction Phase

In the VDM construction phase, NAssim aims to construct a refined and validated native device model based on its user manual. We start by describing the data structure of VDM. Then we describe the key enablers of VDM construction: Parser Framework and Validator.

VDM Data Structure: We design a semantics-enhanced tree structure to represent the native device model. Nodes of tree are CLI command templates with placeholder parameters, *e.g.*, node 'peer <ipv4-address> group <group-name>'. Edges of tree represent configuration hierarchy, *e.g.*, edge from node 'bgp <as-number>' to node 'peer <ipv4-address> group <group-name>' denotes the latter CLI command works under the sub-view enabled by the former. Each node are linked to its relevant semantic corpus extracted from user manual, *e.g.*, node 'peer <ipv4-address> group <group -name>'is associated with corpus shown in Figure 3. The entire tree describes the configuration model of the device, including the CLI command set supported, the command hierarchy and the semantics of commands.

Parser Framework: We develop a parsing framework to address heterogeneous formatting of semantic descriptions of multi-vendor CLI commands and cast them into a vendor-independent format. The format, shown in Figure 3 and Table 3, acts as an unified container to normalize the vendor diversity on manual formats. NetOps engineers can build customized vendor-specific parsers using this framework. The output of a parser is the preliminary VDM.

Validator: Validator addresses the manual ambiguity. It identifies flaws in the preliminary VDM, and generates the validated VDM. Validator examines three aspects of the preliminary VDM to enforce rigorous validation: 1) formal syntax validation to identify ambiguous CLI template, 2) CLI hierarchy derivation and validation, and 3) validation against empirically-verified configuration files from running devices. The output is a validated VDM.

3.2 VDM-UDM Mapping Phase

In this phase, NAssim aims to identify parameter-level relationship between VDM and UDM.

UDM Data Structure: UDM is often stored as a tree hierarchy [16]. Nodes of the UDM tree denote configuration attributes, such as IP address of a interface, name of an ACL policy and *etc.* An attribute may be configured by a specific CLI command parameter in individual VDM, but the attribute and the equivalent parameter can have different names. Sub-trees generally denote a group of relevant attributes, *e.g.*, attributes for a specific network protocol. In the common practice for managing multi-vendor network, UDM covers the common functionalities across all the devices, which is also the intersection across VDMs of multiple vendors. In this work, we assume the UDM is given, and we leave augmenting UDM by building CLI models for proprietary CLI commands of specific vendors as future work. NetOps engineers who construct the UDM usually annotate the UDM with brief context for attributes to facilitate future review and extension [16].

Mapper: Mapper aims to address the model heterogeneity. Even for NetOps experts, the VDM-UDM mapping is tedious and error-prone due to the considerable size of both models. As shown in Table 4, both Huawei and Nokia VDMs contain $>10^4$ nodes. Mapper exploits the semantic context attached with both VDM and UDM, and then leverages recent NLP advances in context encoding and similarity measurement to realize VDM-UDM matching. The output of Mapper is the most potential matched CLIs/parameters with interpret-able semantics for each attribute on the UDM. We design the output of Mapper to be human-comprehensible: for each parameter in a CLI command, Mapper outputs a list of recommended, most related attributes in UDM. NetOps engineers can directly work on the output with their domain knowledge to further confirm the VDM-UDM mapping, and, if the Top-1 recommendation is wrong, they can modify the mapping results. We also collect the expert-corrected mapping results, and we use them as labelled training/testing sets to continuously improve Mapper's NLP models for semantic context encoding and matching, which benefits future VDM-UDM mapping procedures.

In the following sections, we elaborate on the three core components of NAssim.

4 NASSIM PARSER FRAMEWORK

Manual parsing is the first step in SNA. The main goal of our Parser Framework is to extract all essential VDM-relevant information from the manuals, and meanwhile normalize diverse manual styles to a unified format to facilitate vendor-agnostic processing in the subsequent steps. To achieve this goal, we first design a unified format with thorough consideration on expressiveness, interpretability and extendibility. Then, we adopt the TDD methodology to ensure the quality and reliability of the parsing.

Vendor-independent VDM Corpus Format: Following our summaries in Table 1, we define a JSON (JavaScript Object Notation) format to contain the major configuration information. Figure 3 show a sample corpus generated by our parsing framework. The vendor-independent format is organized in dictionary format using attributes in Table 1 as keys. We restrict the type of each fields as shown in Table 3. NAssim's vendor-independent parsed format captures the only commonality of manuals from different vendors, and the it is easy to expend additional information to the JSON. The human-readability of this format also makes it easy for both the NetOps experts and the downstream NLP algorithms.

Keys	Type Restriction
CLIs	a list of string (non-empty list)
FuncDef	string
ParentViews	a list of string (non-empty list)
ParaDef	a list of dict (Keys:"Paras" and "Info")
Examples	a list of list

Table 3: Format Definition of Vendor-Independent Corpus (JSON)

Test-Driven Parser Development: As shown in the § 2.2, there is no trivial way to parse the online version of manuals. To guarantee completeness of parsing, NAssim's parsing framework adopts the Test-Driven Development [29] principle to specify a workflow for building parsers.

The architecture of our parsing framework is shown in Figure 2. `Parser` acts as the base parser class, inherited by its sub-classes `Parser_<vendor>`. In principle, we only need to implement `Parser_<vendor>` for each vendor once. This framework makes it convenient to accommodate a consolidated and extensible testing scheme into the base parser class to benefit its all sub-classes. We list the tests in Appendix B.

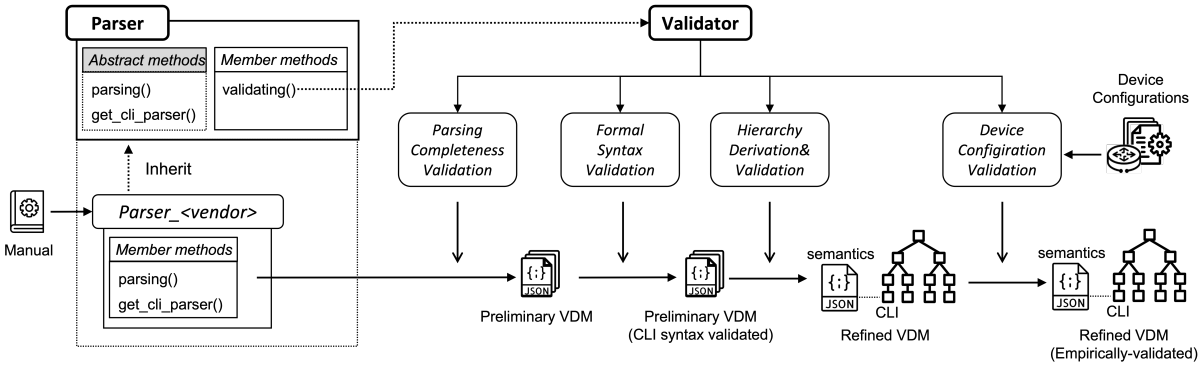


Figure 2: Detailed workflow of Parser Framework and Validator in VDM construction phase.

```
{
 "CLIs": [
   "peer <ipv4-address> group <group-name>",
   "undo peer <ipv4-address> group <group-name>"
 ],
 "FuncDef": "The peer group command adds a peer to a peer. The undo peer group
command deletes a peer from a peer group and all configurations of the peer. By default,
no peer group is created",
 "ParentView": ["BGP view"],
 "ParaDef": [
   {"Parameters": "ipv4-address",
    "Info": "Specifies the IPv4 address of a peer. It is in dotted decimal notation."},
   {"Parameters": "group-name",
    "Info": "Specifies the name of a peer group. The name is a string of 1 to 47 case-
sensitive characters, with spaces not supported."}
 ],
 "Examples": [
   ["<HUAWEI> system-view",
    "[~HUAWEI] bgp 100",
    "[*HUAWEI-bgp] group test internal",
    "[*HUAWEI-bgp] peer 10.1.1.1 group test"]
 ]
}

// Extended corpus with distilled ParaType info
{
 "CLIs": [
   "peer <ipv4-address> group <group-name>",
   "undo peer <ipv4-address> group <group-name>"
 ],
 "...": "... ",
 "ParaType": {
     "ipv4-address": [ipv4],
     "group-name": [string, 1, 47]}
}
```

Figure 3: A sample of parsed VDM corpus.

The workflow to support a new vendor/model in our parsing framework is as follows:

(0) (optional) We augment the base Table 3 with addition type constraints for this vendor; an automated procedure then generates a set of tests;
(1) We sample a batch of manual pages to develop a preliminary version of `Parser_<newvendor>`, which inherits the base `Parser` class and implements the specific `parsing()` method;
(2) We call the `validating()` method implemented in the base `Parser` to verify the completeness of parsed corpus using the generated tests, producing a summary report for violations.
(3) We follow the violations in summary reports to amend the parsing logic if necessary.

We iterate Step 2&3 until the parsed corpora pass all the tests. The reports has two parts: 1) summary of key attributes, which lists all corpus with problematic/empty 'CLIs' fields, and relevant external links to the original manual pages; 2) status of corpus, which lists all problematic fields of each corpus. The developer can sample most problematic corpus/pages to improve the parser.

Since manual parsing is situated at the very upstream of the SDN assimilation, the correctness of parsed corpora are crucial for downstream tasks. The reliability of the TDD-based human-in-the-loop methodology makes it a good choice for our scenario to speed up improving the quality of the parsed corpora in an interpretable way.

5 NASSIM VALIDATOR

In the Parser Framework, we introduce the TDD principle to greatly reduce the information loss due to careless parsing. However, user manuals are human-written documents, and it is inevitable for them to have mistakes and typos. These mis-information in original user manuals always penetrates NAssim's Parser Framework. In order to mitigate their negative impacts on downstream tasks, we design a rigorous validation scheme as shown in Figure 2, including formal syntax validation for CLI commands (*i.e.*, the 'CLIs' field in Figure 3), configuration model hierarchy derivation and validation, and validation with empirical data.

5.1 Formal Syntax Validation

The 'CLIs' field is particularly sensitive to mis-information, because a CLI command with a wrong format can not be accepted by the network device. However, as shown in § 2.2, CLI commands in manuals can contain syntactic errors, and cannot be fully trusted. This motivates us to design a rigorous validation method to systematically audit CLI commands in parsed corpora, so that the experts can intervene in more targeted and efficient way.

The preambles of user manuals provide conventions to illustrate the syntax of their 'CLIs' fields. For example, most manuals use curly braces { } to indicate selected branches, and brackets [ ] to indicate optional branches [3], as shown in Figure 4. We express these command conventions/syntax into their equivalent Backus Normal

[3] https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus5500/sw/command/reference/unicast/n5500-ucast-cr/n5500-ucast-preface.html

Command descriptions use these conventions:

Convention	Description
boldface font	Commands and keywords are in boldface.
italic font	Arguments for which you supply values are in italics.
[ ]	Elements in square brackets are optional.
{x | y | z}	Alternative keywords are grouped in braces and separated by vertical bars.
[x | y | z]	Optional alternative keywords are grouped in brackets and separated by vertical bars.
string	A nonquoted set of characters. Do not use quotation marks around the string or the string will include the quotation marks.

Figure 4: Command convention of Cisco manuals.

```
import pyparsing as p

# syntax parser for Cisco CLI
word = p.Word(p.printables, exclude_chars='{}[]|#\n').setParseAction(leaf_gen)
ele = p.Forward()
items = ele + p.ZeroOrMore('|' + ele)
select = p.Group('{' + items + '}').setParseAction(select_gen)
option = p.Group('[' + items + ']').setParseAction(option_gen)
ele <<= p.OneOrMore(option ^ select ^ word).setParseAction(ele_gen)

syntax_parser = ele
```

Figure 5: Code snippet for syntax parser generation.

Form (BNF) [43], and then transform them into CLI command syntax parsers by leveraging parser generator tools like pyparsing [18] or ANTLR [1]. Figure 5 shows a `pyparsing` snippet to generate a syntax parser complying to command conventions shown in Figure 4. To validate against formal syntax, we can call the syntax parser to parse the 'CLIs' field of corpus in automated way, *i.e.*, check whether they conform to the formal syntax. We record all parsing failure cases to quickly identify problematic CLIs fields in the preliminary VDM corpus. Then the NetOps experts can conduct targeted interventions to correct them so as to improve the VDM corpus credibility.

5.2 Model Hierarchy Derivation and Validation

CLI model hierarchy refers to the relationship between individual CLI commands, *i.e.*, each command has a viable working view, *i.e.*, parent view enabled by its parent CLI command. Configuration models of most vendors follows tree-based hierarchy, but the tree is usually implicit in manuals—only Nokia's manuals specify their hierarchy along with individual CLIs syntax/semantic description. Some devices have special CLI commands to display their model hierarchy on the terminal, but only command syntax is shown without explicit linking to their semantic descriptions.

We find a reliable way to derive model hierarchy accompanied with semantic context by fully exploiting 'Examples' fields in VDM corpus. Revisiting the structure of corpus shown in Figure 3, most fields centred around the 'CLIs' field to provide detailed semantic description. However, the 'Examples' field demonstrates an instantiated version of current 'CLIs' field in a configuration snippet, implicitly bridging the relationship between the current CLI and its parent CLI in the instantiated cases. Therefore, the basic idea of our hierarchy derivation scheme is as follows: take Figure 3 as the example, for a corpus, we first identify the corresponding CLI instance in 'Examples' fields, *i.e.*, `peer 10.1.1.1 group test`. Then we track back to find the parent CLI instance based on the prefix indentation, *i.e.*, `bgp 100`. Finally, we search within all corpus to find a 'CLIs' field matching with CLI instance `bgp 100`, *i.e.*, `bgp <as-number>` in this case. Combined with 'BGP view' indicated in 'ParentView' field, it follows that the CLI command `bgp <as-number>` enters the 'BGP view'. Other 'CLIs' with the same

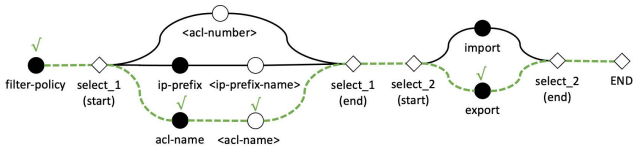


Figure 6: Toy example: match a CLI instance with the graph model of a CLI template.

parent view can directly reuse the previous derivation, or they can derive based on their own 'Examples' fields for majority voting.

CLI Graph Models: Enabling the above hierarchy derivation scheme (the first and third step) requires implementing a basic function: determine whether a CLI instance matches a CLI command template, *i.e.*, the contents in 'CLIs' fields. We design a state-machine-based template matching algorithm for this task. We elaborate the main idea with a toy example. Given a CLI template as follows:

```
filter-policy { <acl-number>
  | ip-prefix <ip-prefix-name>
  | acl-name <acl-name> }  { import | export }
```

To check whether it can be instantiated as a specific CLI instance, *e.g.*, `filter-policy acl-name acl1 export`, we first construct the CLI graph model as shown in Figure 6. A CLI graph model (CGM) is a finite state machine with a single root and a single terminal. Then, we search paths from root node in breadth-first manner to match the tokens in CLI instance. Keyword nodes (*i.e.*, solid circles in Figure 6) in the graph require exact text matching, while parameter nodes (*i.e.*, hollow circles in Figure 6) only require type matching, *e.g.*, `string`, `int` or `ipv4-addr`. If a matched path from root to terminal (*e.g.*, dotted green line in Figure 6) is found, we can match the CLI instance with this template. Due to space limit, we show detailed CGM algorithm with examples in Appendix C.

CLI Instance-Template Matching: With CGM as input, we develop a CLI instance-template matching algorithm to determine whether a CLI instance is matched with a CLI template (cli_{graph}), shown in Algorithm 1. In a nutshell, we conduct breadth-first search to find a path from root to sink to match the tokens of CLI instance. In each search step, we first find all potential candidates for the next element. If all of them are unmatched, the matching algorithm can terminate early. Otherwise, we record matched ones and proceed to next search step.

In practice, the CLI model hierarchy derivation scheme can retrieve the majority of the hierarchy, but does not guarantee a full recovery, because of the potential lack of data in original manuals. For example, in Huawei corpus, we found that the relevant example snippet for 'VPN instance MSDP view' is shared with another view as shown in Figure 7. It is difficult and unreliable for an algorithm to surely determine whether `msdp` command enables the first view or the second or both. Therefore, we introduce an additional validation step after CLI hierarchy derivation. Specifically, we quantify the certainty of the derivation based on the number of views and examples of its data source. For a working view, if we fail to reliably associate it with an example snippet, we record it with all potentially relevant snippets, so that NetOps can review them later to resolve these ambiguous cases with their domain knowledge. With large-scale automated derivation and targeted validation, we can

Algorithm 1: CLI Instance-Template Matching

```
Func is_cli_match(cli, cli_graph):
    cli_eles = cli.split()
    next_ind = 0
    next = cli_eles[next_ind]
    next_candis = get_graph_root(cli_graph)
    while True do
        res = match_next(next, next_candis, cli_graph)
        if not res['match_flag'] then
            break
        next_candis = get_next_candis(cli_graph, res['matched'])
        next_ind += 1
        if next_ind >= len(cli_eles) then
            break
        next = cli_eles[next_ind]
    if next_ind >= len(cli_eles) and is_reach_end(next_candis) then
        return {'match_flag': True}
    return {'match_flag': False}
```

```
{
 "CLIs": [
  "import-source acl { acl-number | acl-name }",
  "undo import-source"
 ],
  "...": "... ",
 "ParentView": ["VPN instance MSDP view", "MSDP view of a public network instance"],
  "...": "... ",
 "Examples": [
  ["<HUAWEI> system-view",
   "[~HUAWEI] acl number 3101",
   "...",
   "[*HUAWEI-acl4-advance-3101] quit",
   "[*HUAWEI] multicast routing-enable",
   "[*HUAWEI] msdp",
   "[*HUAWEI-msdp] import-source acl 3101"]
 ]
}
```

Figure 7: An example ambiguous view not be resolved by our model hierarchy derivation scheme.

obtain a complete CLI model hierarchy with semantic descriptions of individual CLI commands, *i.e.*, the validated VDM.

5.3 Validation with Empirical Data

We so far mainly utilize contents in the parsed VDM corpus for validation. In this part, we take a further step to leverage another data source, empirical device configurations. The workflow of this stage of validation is shown in Figure 8: for each CLI instance in a configuration file, we first find its matched CLI template; then we check whether the matched template and that of its parent CLI instance form a parent-child relationship on the CLI hierarchy. We record unmatched CLI instances and the reasons (*e.g.*, not found matched CLI template, unmatched hierarchy) for the experts to audit later.

We note that the configurations on the existing running-devices may not cover the all CLI commands, since the running devices may only enable needed features. However, CLI commands covered by the current device configurations are the most commonly-used ones in practice.

For those CLI commands which are unused in empirical configurations, we leverage the constructed CGM as shown in Figure 6 to generate CLI command instances (*e.g.*, enumerating paths from root to sink and instantiating the parameter nodes); then we issue the instances directly to the devices for validation; finally, we use 'show' commands (or equivalent commands on non-Cisco devices) to check whether the instance has been correctly configured in the

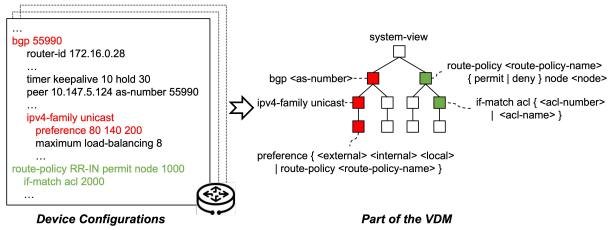


Figure 8: Validation against device configuration.

device. Correct instances then becomes empirical configurations from running devices, and we can run the same workflow above.

Through comprehensive validation schemes, we obtain validated device models in the form of CLI hierarchy and semantic context of individual CLIs, *i.e.*, the validated VDM. This lay a solid foundation for the Mapper.

6 NASSIM MAPPER

Due to the sheer number of CLI commands and parameters, handcrafting such a mapping is tedious and error-prone. Thus we seek to automate this process as much as possible. Using the Parser and Validator, the VDM contains abundant semantic information for all CLI commands and their parameters, and the parameters in the UDM is also enriched with context information. Our key innovation is to let the Mapper "learn" the knowledge from the context information in VDM by applying recent advances in NLP and deep learning (DL), and then predict the most likely mapping to UDM.

Problem Formulation: Essentially, a mapping between two device models is an alignment of the model parameters between the two models. Given a VDM V with n_V parameters $P^V=(p_0^V,p_1^V,\cdots,p_{n_V-1}^V)$, as well as a UDM U with n_U parameters $P^U=(p_0^U,p_1^U,\cdots,p_{n_U-1}^U)$, a mapping $M_{V\bowtie U}$ is a bipartite graph between P^V and P^U.

Mapper Workflow: The architecture of the NAssim Mapper is illustrated in Figure 9. Given a VDM, we first find the relevant VDM corpus for each of the model parameters and then locate the key context information, *e.g.*, descriptions of the CLI command and its parameters. For a UDM, we directly retrieve the context information of each parameter. Then, we use a DL-based model to encode the context information and obtain the vector representations, *i.e.*, the context embedding. Finally, we evaluate if a pair of parameters from two models are referring to the same concept by measuring the similarity between their context embedding vectors. We go through all possible pairs of parameters from the two models and eventually obtain the mapping $M_{V\bowtie U}$. The UDM we use in our prototype is a proprietary model designed by the experts based on their years of experience on managing large enterprise networks. Figure 10 shows an example of mapping a parameter of a VDM to an attribute of our UDM.

6.1 Context Extraction

For each parameter, we extract relevant semantic information from the VDM (or UDM) as its context. Formally, for VDM (or UDM) M , with n_M parameters and one of its parameter p_i^M ($i\in\{0,1,\cdots,n_M-1\}$), we use $c(p_i^M)=[s_i^0,s_i^1,\cdots,s_i^{k_M-1}]$ to represent the

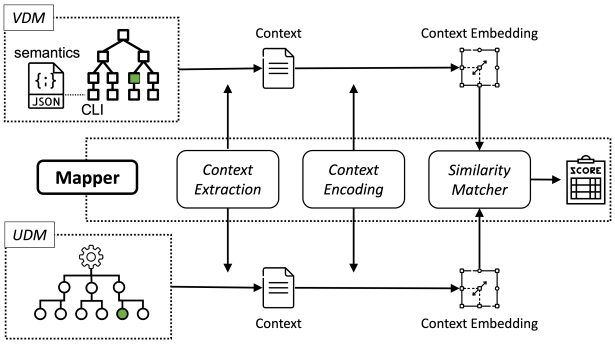


Figure 9: Detailed workflow of `Mapper` in VDM-UDM mapping phase.

context we extract for p_i^M, where each s is a text sequence and k_M is the number of extracted sequences. Depending on the amount of available information of each device model, k_M for each model could be different. Among available contextual information, we find the following ones valuable for the mapping tasks: the name of parameters and CLI commands, the description of parameters, the parent views, and the function description of the CLI commands.

6.2 Context Encoding/Matching with NetBERT

To encode the contextual information into vectors, BERT [32] is the most popular NLP model suitable for this task. Given a sequence of text tokens, BERT produces a contextualized representation of each token with a stack of Transformers [50]. SBERT[45] is a siamese network of BERT, which is pre-trained with the sentence matching objective and a large NLI dataset. It is capable of encoding two semantically similar text sequences into two embedding vectors that are close in the embedding space. However, such a network may have limited performance in a domain that is never seen in the pre-training corpus.

Therefore, we adopt and adapt SBERT for the task of context encoding, and we name the resultant model NetBERT. NetBERT is inherited from an existing pre-trained SBERT model, which is fine-tuned with a sentence matching objective on mapped VDM-UDM parameter pairs to achieve Domain Adaptation (DA).

Embedding Generation: Before going through the details of DA, we show how NetBERT is used to do the context encoding. We use the NetBERT model to encode the text sequences in the contextual information into vector representations. We encode each of the text separately and then produce an embedding matrix, which we call a context embedding. Assume the output dimension of the encoder $e(\cdot)$ is m, then the context embedding of parameter p_i^M is formulated as Equation 1.

$$E_i^M = e(c(p_i^M)) = e([s_i^0, s_i^1, \cdots, s_i^{k_M-1}]) \in R^{k_M \times m} \quad (1)$$

If we want to map a VDM V to a UDM U, and one of the pairs of parameters is (p_i^V, p_j^U). Then, the encoder $e(\cdot)$ produces a pair of context embedding vectors $(E_i^V, E_j^U) \in (R^{k_V \times m}, R^{k_U \times m})$, where k_V is not necessarily equal to k_U.

Parameter Mapping: The mapping between two model parameters can be determined by evaluating the similarities of their corresponding context embedding vectors. We use row-wise cosine similarity to measure the similarity between two context embedding vectors. For each pair of row vectors from $\mathbf{E}_i^V$ and $\mathbf{E}_j^U$, we compute a cosine similarity score. Given the number of contexts k_V and k_U, we have $k_V \times k_U$ pairs of vectors compared. Then, we build a weight vector $\mathbf{w} = \langle w_0, w_1, \cdots, w_{k_V \times k_U - 1} \rangle (s.t. \sum_{t=0}^{k_V \times k_U - 1} w_t = 1)$ to balance all pairs of context groups, and combine all the cosine scores via a weighted sum. Equation 2 shows the process of computing the similarity between $\mathbf{E}_i^V$ and $\mathbf{E}_j^U$. The operator $\otimes$ computes the row-wise cosine similarities of two matrices.

$$Sim(\mathbf{E}_i^V, \mathbf{E}_j^U) = \mathbf{w} \cdot (\mathbf{E}_i^V \otimes \mathbf{E}_j^U) \quad (2)$$

The weight matrix $\mathbf{w}$ is a hyper-parameter, which can be manually specified or automatically generated via grid search. In the simplest setting, we may set $\mathbf{w}$ as a uniform weighting vector, i.e. $w_t = \frac{1}{k_V \times k_U}, \forall w_t \in \mathbf{w}$. Given a VDM parameter, we use the above similarity measurement to find top-K similar parameters in UDM as the mapping recommendations.

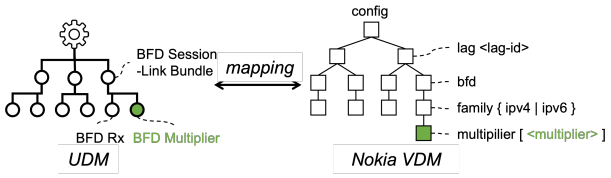


Figure 10: An example of VDM-UDM mapping.

6.3 Fine-tuning NetBERT

Given a few mapped VDM-UDM parameter pairs labeled by NetOps experts, we may generate a training corpus for fine-tuning the NetBERT model. We treat all the mapped pairs as positive pairs and do random sampling to generate the negative pairs. For each pair of parameters, we extract their contexts and feed them as the input into the current NetBERT model. We use exactly the same siamese architecture as the the original SBERT model and the sentence matching training objective to do the fine-tuning. In the case of unsupervised setting, *i.e.*no mapped VDM-UDM is provided, NetBERT is equivalent to SBERT. But with more human efforts involved, NetBERT is able to be more adapted to the network configuration domain.

7 EVALUATION

We implement a prototype of NAssim in Python 3.8. Using this, we evaluate NAssim in the following aspect: 1) the effectiveness and reliability of VDM construction phase; 2) the performance of VDM-UDM mapping phase.

7.1 Prototype Implementation

For the manual parsing framework, we implement the base `Parser` and `Parser_<vendor>` to support parsing online manuals (HTML format) of four popular vendors: Cisco, Huawei, Nokia and H3C. We mainly utilize the Beautiful-soup library [3] to implement their `parsing()` functions, and pyparsing [18] as parser generator to implement the CLI formal syntax parser. We implement the Validator in ~1200 lines of code (LOC), and we spend most effort in model hierarchy derivation and validation (~700 LOC), which utilizes the NetworkX [14] library. For the Mapper, we use the PyTorch [19] library to implement the model and algorithms in

Vendor/Model/ReleaseYear	-	Huawei/NE40E/2021	Cisco/Nexus5500/2011	Nokia/7750SR/2021	H3C/S3600/2009
Main Statistics	#CLI Commands	12874	278	14046	759
	#Views	607	27	3832	28
	#CLI-View Pairs	36274	366	22734	851
Adaption Cost	`parsing()` LOC	45	52	57	41
	`get_cli_parser()` LOC	8	6	10	8
Syntax Validation	#Invalid CLI Commands	13	19	139	13
Model Hierarchy Derivation & Validation	#Example Snippets	15466	523	/	1147
	Construction Time (second)	785.58	14.29	94.56*	34.3
	#Ambiguous Views	47	8	/	4
Device Configuration Validation	#Config Files	197	/	416	/
	Matching Ratio	100%	/	100%	/

Table 4: Evaluation of the VDM Construction Phase. *Nokia manuals do not provide examples, but they explicitly specify model hierarchy in the manuals. Thus, we extract the hierarchy using `Parser_<nokia>` by implementing extra functions

~800 LOC. We evaluate our implementation on Ubuntu 18.04 with a 8-core 3.0GHz CPU, 64GB memory and 1 Nvidia V100 GPU.

7.2 Evaluation of VDM Construction Phase

In this part, we would like to answer the essential question that whether it is feasible and reliable to construct vendor-specific device models from their user manuals. Therefore, we demonstrate our operational experiences and results on applying our proposed Parser and Validator to construct refined and validated VDM from four manuals of popular vendors: Huawei NE40E Command Reference [13], Cisco Nexus 5500 Series NX-OS Unicast Routing Command Reference [4], Nokia 7450 ESS/7750 SR/7950 XRS Reference [15] and H3C S3600 Command Manual [11].

Statistics of Parsed VDMs We first summarize basic statistics of constructed VDMs. Table 4 show the total number of CLI commands in four VDMs. Nowadays, a manual can covers up to 10k+ CLI commands. In our evaluation, we find it common that a single CLI command works under multiple views. For example, `peer <ipv4-address> as-number <as-number>` can work under BGP view, BGP multi-instance view, BGP-VPN instance view *etc.* to create a peer with specified AS-number. In the CLI model hierarchy, we should use multiple nodes to represent the CLI command with different parent CLI commands. In other words, the size of VDM should be quantified by the number of CLI-View pairs. As shown in Table 4, the numbers of CLI-View pairs are prominently larger than that of CLI commands for all vendors. The considerable size of VDMs justifies the need to design an automated framework.

Adapting Parser to New Vendor We next investigate the adaption cost. Considering the major parsing logic is similar across all vendors, we develop a basic code skeleton of `parsing`. For implementing each vendor' parser, we add vendor-specific processing details to the skeleton, mainly modifying content extraction logic of fields in Table 3. Empirically, we use the modified LOC of vendor-specific `parsing()` against basic skeleton, and LOC of `get_cli_parser()` functions to quantify the efforts required for a new vendor. As our evaluation results shown in Table 4, it takes ~50 LOC per vendor, which is an acceptable one-time cost.

Formal Syntax Validation We apply the syntax validation scheme to identify problematic CLI commands in the preliminary VDM. It is unsurprising that we identify invalid CLIs in all vendors, 184 syntactic errors in total as shown in Table 4, because manuals are human-written documents after all. However, it is noted that the ratio of invalid CLI commands over total ones are minimal. We believe that user manuals, as official documents should experience a certain rounds of proofreading and editing. With basic manual quality assurance and revision guidance provided by our validation, the credibility of the derived VDM can be further enhanced.

Model Hierarchy Validation The computation complexity of model hierarchy derivation depends on the number of views and the number of examples, as is shown in Table 4. The largest model hierarchy takes around 13 minutes to construct, with around 84% of processing time is spent on the CLI graph model (CGM) generation for all CLI commands before starting CLI instance-template matching on the example snippets. We believe the cost of time is reasonable as it is only done once per manual. The validator also identifies 59 ambiguous views in four manuals which are then reported to NetOps for revision.

Empirical Data Validation In our evaluation, we only conduct device configuration validation for Huawei and Nokia due to the availability of large numbers of their configuration files from data center network. We collect 613 configuration files in total, 197 for Huawei and 416 for Nokia. Huawei set has 93617 lines of command instances with 17391 unique lines. Nokia set has 163854 lines with 38352 unique ones. We validate the Huawei and Nokia VDMs against corresponding configuration files as illustrated in Figure 8. We find that 100% CLI instances in configuration files can be matched to nodes on the CLI model hierarchy, which confirms the completeness of our hierarchy derivation schemes. In our evaluation, the Huawei dataset is associated with only 153 command templates, which is few compared to the 12874 total templates. This corroborates with the fact that this dataset is from data centers, where the same set of functions are used in thousands of devices.

7.3 Evaluation of VDM-UDM Mapping Phase

We proceed to answer the question: can the proposed Mapper ease the burden of NetOps engineers in SNA? As mentioned before, Mapper acts as a recommender system to generate most likely mapping between parameters of VDM and UDM. Thus, we evaluate

Mapping Setting	Models	k in recall@top k (%)							
		1	3	5	7	9	10	20	30
Huawei-UDM	IR	41	61	69	76	79	80	90	93
	SimCSE	40	59	66	68	70	72	77	81
	SBERT	53	72	79	81	84	85	89	92
	IR+SimCSE	43	68	75	79	81	82	89	92
	IR+SBERT	56	75	81	85	87	88	91	94
	NetBERT	57	74	80	85	86	87	91	94
	IR+NetBERT	**58**	**78**	**83**	**86**	**88**	**89**	**93**	**95**
Nokia-UDM	IR	24	41	48	57	59	60	66	70
	SimCSE	20	31	37	39	39	42	45	48
	SBERT	34	38	49	49	52	52	58	53
	IR+SimCSE	24	35	42	46	48	48	57	61
	IR+SBERT	34	42	52	54	55	58	62	72
	NetBERT	34	43	53	**66**	67	**70**	71	73
	IR+NetBERT	**35**	**47**	**55**	65	**68**	68	**71**	**73**

Table 5: Mapper performance

the mapping performance of Mapper to quantify its benefits to NetOps.

Metric: To evaluate the mapping performance of the Mapper, we adopt Recall@top-k, which denotes the percentage of test cases where the correct matching parameters are in top k recommendation by Mapper. Higher recall at smaller k implies that Mapper is more helpful to assist NetOps.

Ground Truth: To evaluate the `Mapper`, we ask NetOps engineers to help annotate Huawei and Nokia VDMs to the given UDM. In total, we have 381 mapping annotations between UDM and Huawei VDM, and 110 for Nokia.

Fine-tuning: Given the fact that the annotated data is scarce, we use cross-vendor tuning and validation to evaluate the effectiveness of fine-tuning. We fine-tuned NetBERT with annotated parameter pairs from Nokia and then evaluate its performance on Huawei dataset. Similarly, we fine-tuned NetBERT on Huawei dataset and then evaluate it on Nokia dataset. We took 1:10 positive/negative rate in negative sampling. We observe that only 1 epoch of training is necessary as more epochs may easily cause over-fitting.

Compared Models: We compare against these models:

- *IR*: A natural idea is to use an information retrieval (IR) approach for mapping candidates generation. We use TF-IDF [38, 41] to measure similarity of parameter pair between UDM and VDM and report ones with k top scores.
- *SBERT*: SBERT [45] enhances the pre-trained BERT network using siamese and triplet network structures to derive semantically meaningful sentence embeddings that can be compared using cosine-similarity.
- *SimCSE*: SimCSE [35] utilizes contrastive learning objective to regularize pre-trained embeddings to favor sentence similarity measuring task.
- *IR+DL*: This model integrate the IR approach and DL models, where DL can be any DL-based sentence matching model. They first utilize the IR approach for roughly matching top-50 candidates, and then leverage DL models to finely rank the candidates to output the final top k matching ones.

Results: We evaluate recall@top k with k from 1 to 30 of different context comprehension models on VDM-UDM mapping tasks. The results are summarized in Table 5. In unsupervised setting, SBERT performs consistently better than SimCSE. The composite IR+DL models achieve better performance than their IR or DL models. When supervision is provided, fine-tuned NetBERT out-performs all the other models, this is because we adapt the NetBERT model to the domain of network configurations. For Huawei dataset, the improvement of fine-tuning is 1-4 percents across all top-k metrics. For Nokia dataset, the fine-tuning achieves up to 18 percents improvement (recall@10). This is because Huawei dataset has much more training pairs than those from Nokia dataset, thus the fine-tuning on Huawei dataset is more effective. Besides, we observe that NetBERT performs as well as IR+NetBERT and even out-performs it in some recall@k. This suggests that a well-tuned NetBERT model can be sufficient for our task. Regarding the save of human effort, for Huawei, NetOps engineers can find correct parameter-pair within NetBERT's Top-10 recommendations with 89% accuracy. This means, if Mapper is allowed to provide 10 suggestions for parameter-pair matching, NetOps engineers only need to refer to the manual 11% of the time during the mapping phase, resulting in acceleration of the mapping phase by 9.1x.

8 DISCUSSION

8.1 Device Configuration Models

There are two common configuration models of network devices: CLI and YANG/NETCONF. CLI is the entry-level way to configure network devices. Almost all devices in the existing infrastructure support CLI functionality, both the latest and the legacy, and almost all vendors have provided comprehensive description of their CLI commands in their manuals Thus, it is intuitive to configure a network via CLI. YANG/NETCONF is regarded as an advanced way for network configuration. YANG is a data modeling language for the NETCONF configuration management protocol. YANG/NETCONF aims to enable pushing and pulling of configuration data in a more structured manners for network automation. However, it is more difficult to master YANG/NETCONF than CLI due to its higher complexity. The YANG provides descriptions of a network's nodes and their interactions. Each YANG module defines a hierarchy of data that can be used for NETCONF-based operations. Modules can import data from other external modules and include data from submodules. Although the YANG schema is indeed a standard, the data represented in the YANG model vary based on vendor implementation. We found that NetOps experts that master YANG/NETCONF is far less than those that master CLI in our field interview. As shown in the repositories maintained by device vendors [23–25], vendor-specific YANG models are less intuitive as their CLI counterparts elaborated in the manuals [4, 13, 15]. In this work, we adopt CLI-based vendor-specific device models (VDM) as the basis so that the SDN assimilation can cover more vendors and devices, the legacy and new; and NAssim can be adopted more smoothly by more NetOps teams at this stage in order to consistently improve our methods. It is believed that the core 'Parsing-Validating-Mapping' philosophy of NAssim can also be applied to address heterogeneity between vendor-specific YANG models and the unified device model (UDM), but additional efforts should be made to summarize intuitive representation from YANG repositories [23–25].

8.2 Vendor-neutral Reference Frame

For decades, NetOps have been trying to ease the burden of network configuration with increasing complexity of multi-vendor enterprise-level networks. The development of YANG is the most representative effort. YANG was initially standardized in 2010 under RFC 6020 [28] and is currently maintained within the Internet Engineering Task Force (IETF). The IETF YANG tries to create vendor-neutral data models. However, it is still limited in scope with what it can configure between multiple vendors, and widely considered the easiest version of YANG to start with. On top of it, vendors create their own versions of YANG to control features specific to their equipment [23–25]. This makes managing multi-vendor networks difficult since different scripts needed to be created and maintained for different devices. The OpenConfig project [16] is then initiated to create a vendor-neutral reference model interoperable between multiple vendors. Today, some vendors like Cisco support OpenConfig YANG. However, their native data models still provide most configuration coverage, while their supported open models only cover a fraction of features [26]. OpenConfig still has a long way to go for complete interoperability. In retrospect of the development course of YANG, it was understood that hardware vendors persistently introduce more features to reinforce their competitiveness. This certainly helps advance network performance, but also creates further fractures in the ecosystems to make it more difficult to build a vendor-neutral configuration reference frame. NAssim makes initial efforts to assimilate device models of multiple vendors from a pragmatic perspective. However, making our networks toward a more manageable infrastructure requires further considerable efforts from the academia and the community.

8.3 Network Assimilation beyond SDN

The design of NAssim is motivated by practical challenges in network configuration management. Most enterprise-level networks present multi-vendor nature and it is prevalent to have a software-defined controller situated between upper-layer network functions (northbound) and heterogeneous network devices (southbound) in current NetOps practices. The southbound of the SDN controller makes the heterogeneous devices transparent to the northbound. For example, if a network function requires to change autonomous system numbers of the BGP protocol, the controller should executive correct configuration commands to put the change into effect on the targeted devices regardless of their vendors. Currently, it requires significant human efforts to assimilate multi-vendor device configuration models into the central controllers. Thus, NAssim seeks to make this process more efficient and affordable for NetOps.

From a broader perspective, the essential research problem behind network assimilation is to identify semantic equivalence between network devices. NAssim is an initial effort to address this problem in the context of software-defined network centralized configuration management. In a nutshell, NAssim aims to identify commands and parameters with the equivalent configuration effect between multi-vendor device models and the unified device model in the controller. In the envision of network autopilot, more efforts should be made to identify other aspects of semantic equivalence beyond configuration commands, also other network paradigm beyond conventional router/switch network, *i.e.,* semantic interoperability in IoT network [27].

9 RELATED WORKS

To the best of our knowledge, we conduct the first comprehensive study and validation of network device manuals, and NAssim is the first work which aims to simplify the process of introducing new vendors and devices with parsed configuration models from manuals.

Recent work [10, 16, 46–49] proposes the use of centralized UDMs, such as OpenConfig, FBNet for network control and management. Several industrial solutions [12, 17, 21, 40] adopt template-based approaches for configuration generation. Many efforts also aim to develop abstract languages or models to specify configuration in a vendor-neutral fashion [6, 16, 49] or in a user-friendly manner [37]. However, none has consider the problem of parsing a configuration model from the semi-structure user manual, thus for these SDN networks, assimilating a new device requires significant human effort.

Another line of related work is network verification, which conducts analysis in provider networks and enterprise networks by formal methods [30, 31, 34, 36, 51]. Yet, they only work for limited vendors, and do not consider errors and inconsistency in manuals. NAssim can help extend them to other vendors they cannot support easily with the parsed and validated corpus and the mapping model.

Recent NLP techniques adopt deep pre-trained language models to encode a text sequence into a high-dimensional embedding vectors, which carries its semantic meaning. BERT[32] is the most popular pre-trained language model and it has been applied in various NLP tasks. While BERT is good at capturing the semantic information of the training corpus, it usually needs to be fine-tuned to fit the downstream task. SBERT[45] is a siamese BERT network pre-trained specifically for sentence matching, which naturally fits our need. SimCSE[35] is another BERT-based model for sentence matching, which is pre-trained with contrastive learning objectives. We evaluate both SBERT and SimCSE on our mapping task, and find them under-perform NetBERT, our fine-tuned and domain-adapted model.

10 CONCLUSION

The current SNA process is a pain for NetOps engineers, requiring significant human effort to reading and understanding the manuals to build and bridge configuration model of a new device to the UDM in the SDN controller. We build NAssim to assist and accelerate the SNA process. Our solution features a unified parser framework, a rigorous validator and a mapper using the domain-adapted BERT model. NAssim liberates the NetOps engineers by learning directly from manuals to form device models which are comprehensible by both the SDN controller and human experts. Our evaluation shows, we can accelerate the assimilation process by 9.1x. We also release a validated and expert-curated dataset of parsed manual corpus for future research.

Acknowledgements: We thank the anonymous SIGCOMM reviewers for their constructive feedback and suggestions. This work was supported in part by funding from the National Key R&D Program of China 2020YFB1807800, the Research Grants Council of Hong Kong (11209520) and CUHK (4055138, 4937007, 4937008, 5501329, 5501517).

REFERENCES

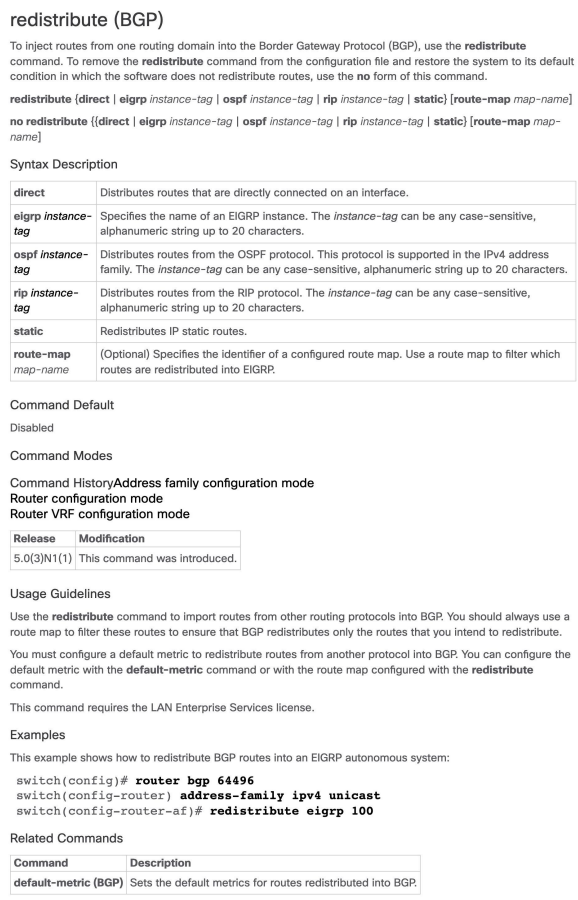
[1] Online; Last accessed Jan. 2022. ANother Tool for Language Recognition. https://www.antlr.org/. (2022).
[2] Online; Last accessed Jan. 2022. Apstra. https://apstra.com/. (2022).
[3] Online; Last accessed Jan. 2022. Beautiful-soup Library. https://beautiful-soup-4.readthedocs.io/en/latest/. (2022).
[4] Online; Last accessed Jan. 2022. Cisco Nexus 5500 Series NX-OS Unicast Routing Command Reference. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus5500/sw/command/reference/unicast/n5500-ucast-cr.html. (2022).
[5] Online; Last accessed Jan. 2022. Cisco sues Huawei over intellectual property. https://www.computerworld.com/article/2578617/cisco-sues-huawei-over-intellectual-property.html. (2022).
[6] Online; Last accessed Jan. 2022. Distributed Management Task Force, Inc. https://www.dmtf.org/. (2022).
[7] Online; Last accessed Jan. 2022. Facebook, Tinder, Instagram suffer widespread issues. https://mashable.com/2015/01/27/facebook-tinder-instagram-issues/. (2022).
[8] Online; Last accessed Jan. 2022. gNXI Tools - gRPC Network Management/Operations Interface Tools. https://github.com/google/gnxi/. (2022).
[9] Online; Last accessed Jan. 2022. Google routing blunder sent Japan's Internet dark on Friday. https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/. (2022).
[10] Online; Last accessed Jan. 2022. Graph-Based Live Queries in AOS. https://apstra.com/products/. (2022).
[11] Online; Last accessed Jan. 2022. H3C S3600 Command Manual. http://www.h3c.com/en/Support/Resource_Center/HK/Switches/H3C_S3600/H3C_S3600_Series_Switches/Technical_Documents/Command/Command/H3C_S3600_CM-Release_1602(V1.02)/. (2022).
[12] Online; Last accessed Jan. 2022. HPE Network Management (HP OpenView). https://www8.hp.com/us/en/solutions/business-solutions/printingsolutions/overview.html. (2022).
[13] Online; Last accessed Jan. 2022. Huawei NE40E Command Reference. https://support.huawei.com/enterprise/en/routers/ne40e-pid-15837?category=reference-guides. (2022).
[14] Online; Last accessed Jan. 2022. NetworkX Library. https://networkx.org/,. (2022).
[15] Online; Last accessed Jan. 2022. Nokia 7450 ESS/7750 SR/7950 XRS Reference. https://infocenter.nokia.com/public/7750SR140R4/index.jsp. (2022).
[16] Online; Last accessed Jan. 2022. OpenConfig. http://openconfig.net/. (2022).
[17] Online; Last accessed Jan. 2022. Opsware. http://www.opsware.com/. (2022).
[18] Online; Last accessed Jan. 2022. pyparsing module. https://pyparsing-docs.readthedocs.io/en/latest/index.html. (2022).
[19] Online; Last accessed Jan. 2022. PyTorch Library. https://pytorch.org/. (2022).
[20] Online; Last accessed Jan. 2022. Stock trading closed on NYSE after glitch caused major outage. https://www.theguardian.com/business/live/2015/jul/08/new-york-stock-exchange-wall-street. (2022).
[21] Online; Last accessed Jan. 2022. Tivoli Netcool Configuration Manager. http://ibm.com/software/products/en/tivonetcconfmana. (2022).
[22] Online; Last accessed Jan. 2022. United Airlines jets grounded by computer router glitch. https://www.bbc.com/news/technology-33449693. (2022).
[23] Online; Last accessed June. 2022. Cisco YANG Model Repository. https://github.com/YangModels/yang/tree/main/vendor/cisco. (2022).
[24] Online; Last accessed June. 2022. Huawei YANG Model Repository. https://github.com/Huawei/yang. (2022).
[25] Online; Last accessed June. 2022. Nokia YANG Model Repository. https://github.com/nokia/7x50_YangModels. (2022).
[26] Online; Last accessed June. 2022. OpenConfig Support in Cisco. https://www.ciscolive.com/c/dam/r/ciscolive/us/docs/2019/pdf/DEVNET-1775.pdf. (2022).
[27] Online; Last accessed June. 2022. What is semantic interoperability in IoT and why is it important? https://www.ericsson.com/en/blog/2020/7/semantic-interoperability-in-iot. (2022).
[28] Online; Last accessed June. 2022. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). https://datatracker.ietf.org/doc/html/rfc6020. (2022).
[29] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.
[30] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proc. ACM SIGCOMM*.
[31] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proc. ACM SIGCOMM*.
[32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423
[33] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *Proc. USENIX NSDI*.
[34] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Proc USENIX NSDI*.
[35] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. (2021).
[36] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. 2015. Management Plane Analytics. In *Proc. ACM IMC*.
[37] Arthur S. Jacobs, Ricardo J. Pfitscher, Rafael H. Ribeiro, Ronaldo A. Ferreira, Lisandro Z. Granville, Walter Willinger, and Sanjay G. Rao. 2021. Hey, Lumi! Using Natural Language for Intent-Based Network Management. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 625–639. https://www.usenix.org/conference/atc21/presentation/jacobs
[38] Karen Spärck Jones. 2004. A statistical interpretation of term specificity and its application in retrieval. *J. Documentation* 60 (2004), 493–502.
[39] Hyojoon Kim and Nick Feamster. 2013. Improving network management with software defined networking. *IEEE Communications Magazine* 51, 2 (2013), 114–119.
[40] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. NetCraft: Automatic Life Cycle Management of Network Configurations. In *Proc. ACM SelfDN*.
[41] Hans Peter Luhn. 1957. A Statistical Approach to Mechanized Encoding and Searching of Literary Information. *IBM J. Res. Dev.* 1 (1957), 309–317.
[42] Ratul Mahajan, David Wetherall, and Tom Anderson. 2002. Understanding BGP Misconfiguration. In *Proc. ACM SIGCOMM*.
[43] Daniel D McCracken and Edwin D Reilly. 2003. Backus-naur form (bnf). In *Encyclopedia of Computer Science*. 129–131.
[44] Juniper Networks. Technical report, May 2018. Whats Behind Network Downtime? Proactive Steps to Reduce Human Error and Improve Availability of Networks. (Technical report, May 2018).
[45] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP*. Association for Computational Linguistics, Hong Kong, China, 3982–3992. https://doi.org/10.18653/v1/D19-1410
[46] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. 2015. Condor: Better Topologies Through Declarative Design. In *Proc. ACM SIGCOMM*.
[47] Xin Sun and Geoffrey G Xie. 2013. Minimizing Network Complexity through Integrated Top-Down Design. In *Proc. ACM CoNEXT*.
[48] Yu-Wei Eric Sung, Xin Sun, Sanjay G Rao, Geoffrey G Xie, and David A Maltz. 2010. Towards Systematic Design of Enterprise Networks. *IEEE/ACM Transactions On Networking* 19, 3 (2010), 695–708.
[49] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proc. ACM SIGCOMM*.
[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *NIPS (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.
[51] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *Proc. ACM OOPSLA*.
[52] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. 2014. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials* 17, 1 (2014), 27–51.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A MANUAL SNAPSHOTS

We show the screen snapshots of the online version of manaual pages from Cisco, Nokia, Huawei, and H3C in Figure 11,12,13,14, respectively.



redistribute (BGP)

To inject routes from one routing domain into the Border Gateway Protocol (BGP), use the **redistribute** command. To remove the **redistribute** command from the configuration file and restore the system to its default condition in which the software does not redistribute routes, use the **no** form of this command.

redistribute {**direct** | **eigrp** *instance-tag* | **ospf** *instance-tag* | **rip** *instance-tag* | **static**} [**route-map** *map-name*]

no redistribute {{**direct** | **eigrp** *instance-tag* | **ospf** *instance-tag* | **rip** *instance-tag* | **static**} [**route-map** *map-name*]

Syntax Description

direct	Distributes routes that are directly connected on an interface.
eigrp *instance-tag*	Specifies the name of an EIGRP instance. The *instance-tag* can be any case-sensitive, alphanumeric string up to 20 characters.
ospf *instance-tag*	Distributes routes from the OSPF protocol. This protocol is supported in the IPv4 address family. The *instance-tag* can be any case-sensitive, alphanumeric string up to 20 characters.
rip *instance-tag*	Distributes routes from the RIP protocol. The *instance-tag* can be any case-sensitive, alphanumeric string up to 20 characters.
static	Redistributes IP static routes.
route-map *map-name*	(Optional) Specifies the identifier of a configured route map. Use a route map to filter which routes are redistributed into EIGRP.

Command Default

Disabled

Command Modes

Command HistoryAddress family configuration mode
Router configuration mode
Router VRF configuration mode

Release	Modification
5.0(3)N1(1)	This command was introduced.

Usage Guidelines

Use the **redistribute** command to import routes from other routing protocols into BGP. You should always use a route map to filter these routes to ensure that BGP redistributes only the routes that you intend to redistribute.

You must configure a default metric to redistribute routes from another protocol into BGP. You can configure the default metric with the **default-metric** command or with the route map configured with the **redistribute** command.

This command requires the LAN Enterprise Services license.

Examples

This example shows how to redistribute BGP routes into an EIGRP autonomous system:

```
switch(config)# router bgp 64496
switch(config-router) address-family ipv4 unicast
switch(config-router-af)# redistribute eigrp 100
```

Related Commands

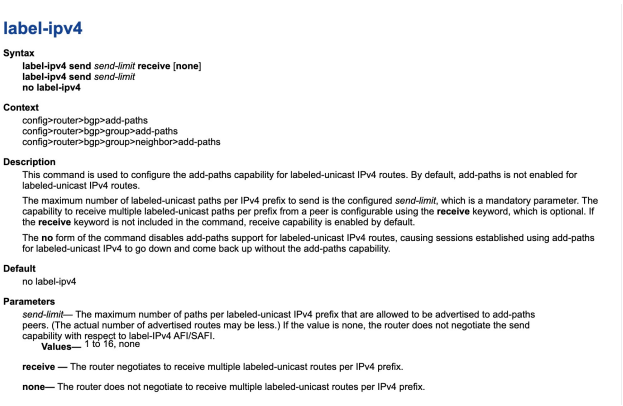
Command	Description
default-metric (BGP)	Sets the default metrics for routes redistributed into BGP.

Figure 11: A Snapshot of Cisco Manual.

B TESTS FOR PARSING COMPLETENESS VALIDATION

In our practice, we specifically enforce the following validation tests to ensure parsing quality.

- *Keys Completeness Test*: the parsed JSON files should be a dictionary with at-least five basic keys in Table 3: 'CLIs', 'FuncDef', 'ParentViews', 'ParaDef' and 'Examples'.
- *Type Restriction Test*: Each fields should comply the type restriction defined in Table 3.
- *CLI Keyword/Parameter Self-check Test*: We enforce an extra check on the critical 'CLIs' fields to make sure the correctness of CLI keywords/parameters identification. In original HTML



label-ipv4

Syntax
label-ipv4 send *send-limit* **receive** [**none**]
label-ipv4 send *send-limit*
no label-ipv4

Context
config>router>bgp>add-paths
config>router>bgp>group>add-paths
config>router>bgp>group>neighbor>add-paths

Description
This command is used to configure the add-paths capability for labeled-unicast IPv4 routes. By default, add-paths is not enabled for labeled-unicast IPv4 routes.

The maximum number of labeled-unicast paths per IPv4 prefix to send is the configured *send-limit*, which is a mandatory parameter. The capability to receive multiple labeled-unicast paths per prefix from a peer is configurable using the **receive** keyword, which is optional. If the **receive** keyword is not included in the command, receive capability is enabled by default.

The **no** form of the command disables add-paths support for labeled-unicast IPv4 routes, causing sessions established using add-paths for labeled-unicast IPv4 to go down and come back up without the add-paths capability.

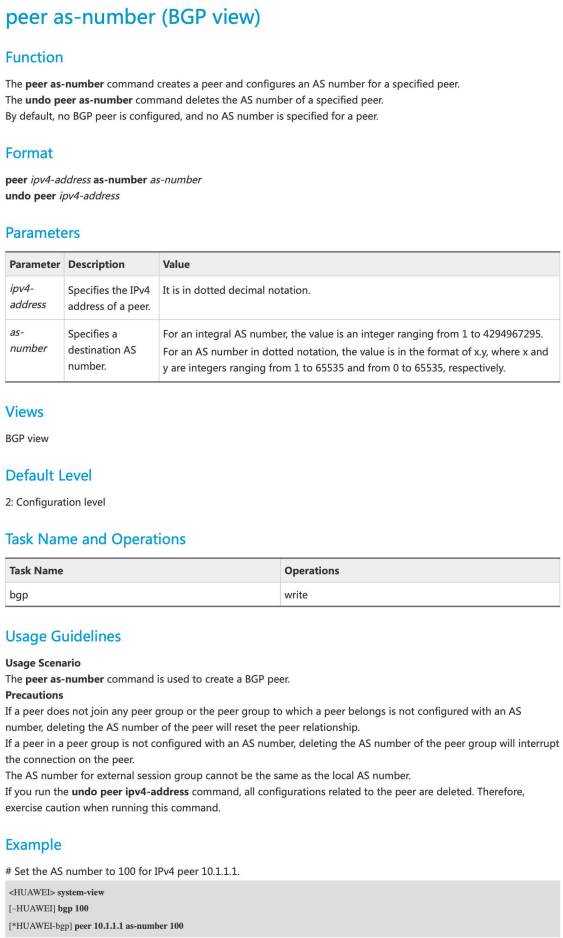
Default
no label-ipv4

Parameters
send-limit— The maximum number of paths per labeled-unicast IPv4 prefix that are allowed to be advertised to add-paths peers. (The actual number of advertised routes may be less.) If the value is none, the router does not negotiate the send capability with respect to label-IPv4 AFI/SAFI.
Values— 1 to 16, none

receive — The router negotiates to receive multiple labeled-unicast routes per IPv4 prefix.

none— The router does not negotiate to receive multiple labeled-unicast routes per IPv4 prefix.

Figure 12: A Snapshot of Nokia Manual.



peer as-number (BGP view)

Function

The **peer as-number** command creates a peer and configures an AS number for a specified peer.
The **undo peer as-number** command deletes the AS number of a specified peer.
By default, no BGP peer is configured, and no AS number is specified for a peer.

Format

peer *ipv4-address* **as-number** *as-number*
undo peer *ipv4-address*

Parameters

Parameter	Description	Value
ipv4-address	Specifies the IPv4 address of a peer.	It is in dotted decimal notation.
as-number	Specifies a destination AS number.	For an integral AS number, the value is an integer ranging from 1 to 4294967295. For an AS number in dotted notation, the value is in the format of x.y, where x and y are integers ranging from 1 to 65535 and from 0 to 65535, respectively.

Views

BGP view

Default Level

2: Configuration level

Task Name and Operations

Task Name	Operations
bgp	write

Usage Guidelines

Usage Scenario
The **peer as-number** command is used to create a BGP peer.
Precautions
If a peer does not join any peer group or the peer group to which a peer belongs is not configured with an AS number, deleting the AS number of the peer will reset the peer relationship.
If a peer in a peer group is not configured with an AS number, deleting the AS number of the peer group will interrupt the connection on the peer.
The AS number for external session group cannot be the same as the local AS number.
If you run the **undo peer ipv4-address** command, all configurations related to the peer are deleted. Therefore, exercise caution when running this command.

Example

Set the AS number to 100 for IPv4 peer 10.1.1.1.

```
<HUAWEI> system-view
[~HUAWEI] bgp 100
[*HUAWEI-bgp] peer 10.1.1.1 as-number 100
```

Figure 13: A Snapshot of Huawei Manual.

pages with rich text format (RTF), keyword and parameters are generally differentiated by their font format. Through our parsing framework `Parser`, parameters should be indicated by angle brackets in plain text as shown in Figure 3.

ospf cost

Syntax

ospf cost *value*

undo ospf cost

View

Interface view

Parameters

value: Cost for running an OSPF process on an interface, in the range of 1 to 65535.

Description

Use the ospf cost command to configure the OSPF cost on an interface.

Use the undo ospf cost command to restore the default.

By default, the OSPF cost on an interface is 10.

You can use the display ospf brief command to display the OSPF cost information.

Related commands: display ospf brief.

Examples

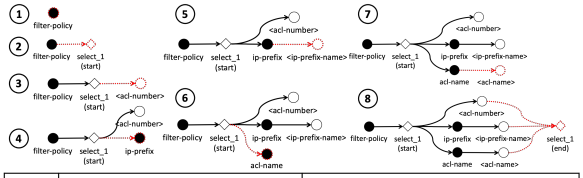
Specify the OSPF cost on the interface as 33.

```
<Sysname> system-view
System View: return to User View with Ctrl+Z.
[Sysname] interface Vlan-interface 10
[Sysname-Vlan-interface10] ospf cost 33
```

Figure 14: A Snapshot of H3C Manual.

To conduct CLI Keyword/Parameter self-check test, we extract the parameter tokens, *i.e.*, those with angle brackets in 'CLIs' fields, and then do a cross-check via comparing these tokens with parameters in 'ParaDef' fields. Through our validation tests, it is quickly found that the Cisco manual interchangeably use 'cKeyword', 'cBold' and 'cCN_CmdName' CSS tags to indicate keywords in CLI commands, and the Huawei manual interchangeably use 'cmdname' and 'strong'. With the help of the test-driven development, we can in general develop a specific `Parser_<vendor>` within one day.

C CGM ALGORITHM DETAILS



State	Before insertion		After insertion	
	prev_stack	tail_stack	prev_stack	tail_stack
1	[ ]	[ ]	[filter-policy]	[filter-policy]
2	[filter-policy]	[filter-policy]	[select_1_start]	[select_1_start]
3	[select_1_start]	[select_1_start]	[<acl-number>]	[select_1_start, <acl-number>]
4	[select_1_start]	[select_1_start, <acl-number>, |,]	[ip-prefix]	[select_1_start, <acl-number>, |, ip-prefix]
5	[ip-prefix]	[select_1_start, <acl-number>, |, ip-prefix]	[<ip-prefix-name>]	[select_1_start, <acl-number>, |, <ip-prefix-name>]
6	[select_1_start]	[select_1_start, <acl-number>, |, <ip-prefix-name>, |]	[acl-name]	[select_1_start, <acl-number>, |, <ip-prefix-name>, |, acl-name]
7	[acl-name]	[select_1_start, <acl-number>, #, <ip-prefix-name>, #, acl-name]	[<acl-name>]	[select_1_start, <acl-number>, |, <ip-prefix-name>, |, <acl-name>]
8	[<acl-number>, <ip-prefix-name>, <acl-name>]	[select_1_start, <acl-number>, |, <ip-prefix-name>, |, <acl-name>]	[select_1_end]	[select_1_end]

Figure 15: An example for CLI Graph Construction.

CLI Graph Model Construction: To construct the CGM, we leverage the syntax parser in the formal syntax validation and the parse action function supported by the `pyparsing` to assist graph model construction, which allows us to specify functions to call after successful matching of the tokens. Thus, we define the following parse action functions:

```
def leaf_gen(tokens):
    return [["leaf", {"name", tokens[0]}]]
def select_gen(tokens):
    return [["select", tokens[0].asList()]]
def option_gen(tokens):
    return [["option", tokens[0].asList()]]
def ele_gen(tokens):
    if tokens.asList() != []:
        return [["ele", tokens.asList()]]
    return tokens
```

The syntax parser as shown in Figure 5 with above parse actions can turn a flat string of CLI template to a nested structure, denoted as cli_{struc}. Take above `filter-policy` command as an example, it can be transformed into the structure shown in Figure 16.

```
['ele', [
 ['leaf', {'name': 'filter-policy'}],
 ['select',[
  '{',
  ['ele', [['leaf', {'name': '<acl-number>'}]]],
  '|',
  ['ele',[['leaf', {'name': 'ip-prefix'}]],['leaf', {'name': '<ip-prefix-name>'}]]],
  '|',
  ['ele',[['leaf', {'name': 'acl-name'}], ['leaf', {'name': '<acl-name>'}]]],
  '}']],
 ['select',[
  '{',
  ['ele', [['leaf', {'name': 'import'}]]],
  '|',
  ['ele', [['leaf', {'name': 'export'}]]],
  '}']]
 ]]
```

Figure 16: A sample of nested CLI structure.

Algorithm 2: CLI Graph Model Construction

```
1 Func get_syntax_graph(cli_struc, cli_graph, dict):
2     foreach ele∈cli_struc do
3         if is_symbol_or_leaf(ele) then
4             symbol_leaf_process(ele, cli_graph, dict)
5         else
6             dict_ori = record_infos(dict)
7             get_syntax_graph(ele[1], cli_graph, dict)
8             post_process(dict_ori, dict)
```

With cli_{struc} as input, we develop a recursive algorithm to construct CLI graph as in Figure 6. The skeleton of recursive CGM construction algorithm are shown in Algorithm 2. Nodes of CGM cli_{graph} come from either leaf or symbols like brackets and braces in the nested cli_{struc}. Thus, we perform graph node/edge insertion in line 4. For nested 'ele', we recursively call the `get_syntax_graph` (line 7) to process the inner elements. However, we record/process the auxiliary information in `dict` (line 6 and 8) before/after recursive calls. Structure `dict` mainly maintains `prev_stack` and `tail_stack` to assist node/edge insertion. For each node, we add directional edges from nodes in `prev_stack` to the current node. The processed nodes that potentially have more children are stored

Algorithm 3: Function Details of CLI Graph Model Construction

```
Func symbol_leaf_process(ele, cli_graph, dict):
    if is_sep_symbol(ele) then
        dict['tail_stack'].append('#')
        return
    if is_start_symbol(ele) then
        node = '_'.join(dict['labels'][ele], len(dict['syms']), 'start')
        dict['sym_stack'].append(node)
    if is_end_symbol(ele) then
        start_node = dict['sym_stack'].pop()
        node = start_node.replace('start', 'end')
        if is_option(node) then
            cli_graph.add_edge(start_node, node)
    if is_leaf(ele) then
        node = ele
    cli_graph.add_node(node)
    foreach prev ∈ dict['prev_stack'] do
        cli_graph.add_edge(prev, node)
    dict['prev_stack'].clear()
    dict['prev_stack'].append(node)
    if is_tail_replace(dict['tail_stack']) then
        dict['tail_stack'][-1] = node
    else
        dict['tail_stack'].append(node)
    if is_end_symbol(ele) then
        reshape_tail_stack(dict)
Func reshape_tail_stack(dict):
    tail_stack = dict['tail_stack']
    end_node = tail_stack[-1]
    start_node = end_node.replace('end', 'start')
    start_ind = tail_stack.index(start_node)
    tail_new = tail_stack[0:start_ind] + [end_node]
    dict['tail_stack'] = tail_new
Func record_infos(dict):
    return copy(dict)
Func post_process(dict_ori, dict, ele, cli_struc):
    next_ele = cli_struc(ele_idx + 1)
    prev_stack_ori = dict_ori['prev_stack']
    if is_sep_symbol(next_ele) then
        dict['prev_stack'] = copy(prev_stack_ori)
    else
        tail_stack = dict['tail_stack']
        last_prev = prev_stack_ori[-1]
        if last_prev ∈ tail_stack then
            ind = tail_stack.index(last_prev) + 1
            dict['prev_stack']= tail_stack[ind:]
```

in the `tail_stack`. When exiting a recursive call on an 'ele' , we can determine next `prev_stack` based on two stacks in `dict_ori` and `dict`. Figure 15 demonstrate the first eight node/edge insertion for constructing CLI graph in Figure 6. The detailed CGM algorithms are shown in Algorithm 2 and 3.

Algorithm 4: Function Details of CLI Instance-Template Matching

```
Func match_next(next, next_candis, cli_graph):
    match_flag = False
    match_states = []
    foreach candi ∈ next_candis do
        if candi is None then
            continue
        if is_keyword(cli_graph,candi) and next == candi then
            match_flag = True
            match_states.append(candi)
    if match_states != [] then
        return {'match_flag': match_flag, 'match_states':
          match_states}
    foreach candi ∈ next_candis do
        if candi is None then
            continue
        if is_para(cli_graph,candi) and is_type_fit(next, candi)
          then
            match_flag = True
            match_states.append(candi)
    return {'match_flag': False, 'match_states': match_states}
Func get_next_candis(match_states, cli_graph):
    next_states = []
    foreach item ∈ match_states do
        succs = cli_graph.successors(item)
        if succs == [] then
            next_states.extend([None])
        foreach succ ∈ succs do
            next_states.extend(get_valid_succs(succ, cli_graph,
              []))
        return set(next_states)
Func get_valid_succssors(node, cli_graph, visited):
    if is_valid_node(cli_graph, node) then
        return [node]
    if cli_graph.successors(node) == [] then
        return [None]
    valid_succ = []
    foreach succ ∈ cli_graph.successors(node) do
        if succ ∉ visited then
            visited.append(succ)
            valid_succ.extend(get_valid_succssors(succ,
              cli_graph, visited))
```

D DETAILED EVALUATION OF NETBERT

We show the detailed evaluation results of mapping performance of Mapper, in addition to § 7.3. We add another metric: mean reciprocal rank (MRR). The reciprocal rank of a list of recommended parameters is the multiplicative inverse of the rank of the first correct answer. The mean reciprocal rank is the average of the reciprocal ranks of all test cases. Higher MRR implies that the recommended parameters of `Mapper` are more accurate.

Mapping Setting	Models	k in recall@top k (%)												MRR
		1	2	3	4	5	6	7	8	9	10	20	30	
Huawei-UDM	IR	41	52	61	66	69	74	76	78	79	80	90	93	0.5401
	SimCSE	40	53	59	63	66	67	68	69	70	72	77	81	0.5148
	SBERT	53	66	72	76	79	80	81	82	84	85	89	92	0.643
	IR+SimCSE	43	61	68	74	75	77	79	80	81	82	89	92	0.5757
	IR+SBERT	56	69	75	79	81	83	85	86	87	88	91	94	0.6737
	NetBERT	57	69	74	78	80	84	85	86	86	87	91	94	0.6732
	IR+NetBERT	**58**	**71**	**78**	**81**	**83**	**85**	**86**	**87**	**88**	**89**	**93**	**95**	**0.6916**
Nokia-UDM	IR	24	31	41	45	48	56	57	59	59	60	66	70	0.3498
	SimCSE	20	27	31	33	37	38	39	39	39	42	45	48	0.2679
	SBERT	34	35	38	44	49	49	49	52	52	52	58	53	0.3908
	IR+SimCSE	24	31	35	40	42	43	46	48	48	48	57	61	0.3241
	IR+SBERT	34	40	42	49	52	52	54	55	55	58	62	72	0.417
	NetBERT	34	40	43	50	53	**58**	**66**	67	67	**70**	71	73	0.4322
	IR+NetBERT	**35**	**41**	**47**	**51**	**55**	57	65	**67**	**68**	68	**71**	**73**	**0.4407**

Table 6: Mapper performance