Delft University of Technology
Master of Science Thesis in Electrical Engineering

Intent-Based Networking for Non-programmable Networks

Dheeraj Ravi

Embedded
Networked
Systems

TU Delft Delft University of Technology

Intent-Based Networking for Non-programmable Networks

Master of Science Thesis in Electrical Engineering

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Dheeraj Ravi
d.ravi@student.tudelft.nl
dheeraj.ravi.92@gmail.com

15-September-2022

Author
Dheeraj Ravi (d.ravi@student.tudelft.nl)
(dheeraj.ravi.92@gmail.com)
Title
Intent-Based Networking for Non-programmable Networks
MSc Presentation Date
15-September-2022

Graduation Committee

Prof. dr. ir. F.A. Kuipers	Delft University of Technology
Dr. J. E. A. P. Decouchant	Delft University of Technology
Mr. Mauro Antonio Di Francesco	Viasat Netherlands B.V.

Abstract

Intent-based Networking (IBN) is one of the hot topics of research in the modern field of networking. Abstracting the complexity of network management away from the network operator through automation is the cornerstone of the IBN concept. However, a lot of current research on intent-based networking is concentrated towards programmable software defined networks (SDN), rather than traditional non-programmable network devices which still hold a large market share in modern networks. Moreover, when it comes to traditional network devices, network validation becomes very crucial as it needs a vendor-agnostic environment to evaluate the network. This thesis studies the important aspects necessary for IBN adaptation for legacy devices and provides a solution for adaptation into modern networks, while being vendor-agnostic. Based on the design, the results obtained from the proofs-of-concept are then analyzed and concluded upon, ending by elucidating avenues of future work.

Keywords - Intent-based networking, traditional network devices, Network validation, vendor-agnostic, OpenConfig

"It is our choices that show what we truly are, far more than our abilities..." –
Albus Dumbledore

Preface

Being a disciple of science since my childhood, there are very few things more life-defining than a Master's degree in a field of my liking. The hunger to learn how things work – and break them apart to understand more – has been my way of life since as long as I can remember. That eventually developed into a love for electronics and telecommunication, which has ultimately led me here to the brink of my Masters in the same field. The world turned upside down in early 2020 due to COVID-19 and will probably never be the same again. This mini-era of human history became a central theme of almost the entire duration of my study here in the Netherlands. It became paramount to constantly remind myself of the higher goal amidst all the uncertainty, hardships, challenges and losses. All of this would not have been possible without the truly special people who supported me throughout this period. This report, in short, represents the blood, sweat and tears of the past 3 years of my Masters at Delft University of Technology and one which I will look back with pride from the moment I graduate.

First, I would like to offer my undying gratitude to *Professor Fernando* - for not just being a great advisor and mentor, but also for providing me an opportunity in the industry with Viasat under such testing times. Without his patient guidance and knowledge, this thesis would not have been possible. My future here in the Netherlands is also largely indebted to him.

Second, to the amazing people at Viasat - *Mauro, Francesca and Mike* - who gave me an opportunity to work on an exciting topic and eventually a career as well. Highly understanding human beings, buddies and in general, great to work with.

Third, to my mother, *Meera* and father, *Ravi* - who sacrificed a lot to help me pursue my dreams and the unconditional support they continue to give me. I cannot thank them enough in one lifetime.

Last, and definitely not the least, to my beloved wife *Sandra* - for tolerating my tantrums, being a partner in crime and a fellow Master student. We undertook a difficult journey together and went through every hardship the last couple of years. Quite simply, the person who has my user's manual.

Dheeraj Ravi
Delft, 5th September 2022

Contents

Chapter 1

Introduction

1.1 Background

Modern society is now almost entirely dependent on the internet. Everything, from daily essentials to the world economy, relies on the internet to function properly. Human interaction is now heavily influenced through social media and exchange of information through various types of digital devices. A fundamental requirement for the usage of such large number of applications is to transfer huge amounts of data at great speeds across the globe. This is done by interconnecting tens of thousands of network devices throughout the world, which exchange information with each other and strive to deliver data from source to destination in milliseconds. These devices all belong to individual Internet Service Providers (or network operators) whose primary job is to ensure proper management of their networks while minimizing costs.

Due to the wide variety of network traffic (data, voice, media streaming, etc.) that needs to be handled, network providers need to offer highly customized and tailor-made services based on the traffic. This is necessary to optimize the use of network resources while ensuring customer service level agreements are fulfilled. Catering to such high requirements also creates high demand on the management of the network and as such, network engineers are tasked with the planning, design, operation and maintenance of the network while trying to keep the operating expenditure as low as possible. Traditionally, engineers manually maintain a database, login to devices and make the necessary changes needed to run the network. A similar approach is followed when there is a fault in the network and there is a need to troubleshoot it. To address this, recent advances in Intent-based Networking (IBN) have made management of networks much easier to network operators and engineers.

The goal of Intent-based Networking is to reduce the complexity of planning, design, operation and management of networks through automation of activities. It involves abstraction of the network from the engineer's point of view by taking inputs known as *'intents'*, while transferring the responsibility of handling and managing the network to a centralized network controller [17]. Intents are generally business or system level policies, which are agnostic to device vendor

and network features, defined to specify the high-level requirements that the network engineer wants to satisfy [13]. Since intents are high-level requirements engineers can specify them without considering the exact technical specifics and how they are implemented, i.e., intents are concentrated more on describing the outcome rather than the process towards the outcome: intents describe *what* the engineer wants and not *how* it is realized [13]. These high-level intents are then translated to lower level policies by the network controller so that they can be applied to the network devices to achieve the desired objective.

A lot of current research on Intent-based networking is concentrated towards programmable software defined networks (SDN) [32][31][5][28][33], since it is easier and faster to apply translated policies on-the-go to network devices. But SDNs are still in the growing phase in the market as network providers migrate services to software based networks [11][21]. This means that traditional, non-programmable network devices such as routers, switches, firewalls etc., continue to be relevant today in almost all modern networks. Therefore, there is a need for an adaptation of intent-based networking towards legacy devices so that network operators can already start building their IBN infrastructures. Moreover, when it comes to traditional network devices, network validation becomes very crucial as explained in later chapters. This thesis studies important aspects necessary for IBN adaptation for legacy devices and attempts to provide a solution for adaptation into modern networks.

1.2 Problem definition

As stated earlier, research and development on Intent-based networking is concentrated more towards software defined networks. But legacy devices still continue to be relevant in modern networks as many services are still configured on traditionally function-specific network devices (like routers, switches, firewalls etc.) through their own vendor-proprietary command line interfaces (CLI). In addition to this, even Virtual Network Functions (NFV), designed to operate on abstracted cloud infrastructure, use vendor specific CLI to configure functionalities and services.

Currently there is a lack of research on the adaptation of IBN for legacy devices. This also causes an additional problem when traditional devices and software-defined networks need to be used seamlessly in a single IBN infrastructure. This thesis aims to address two important aspects of this adaptation of legacy devices to an IBN framework - first, to provide a common solution which is vendor-agnostic so that intents can be translated to any dedicated network device and second, to design a framework for pre-validating network intents for these legacy devices, before changes are applied on a live network.

1.3 Research Questions

The following questions will be central to the research done in this thesis:

1. **RQ1** How to design a framework which helps adapt the IBN concept to traditional, non-programmable network devices. This framework should be seamless for all types of network devices and be vendor-agnostic.

2. **RQ2** How to design an intent structure for low-level intents (defined in section 2.1 which can be used to pre-validate two popular types of network requirements namely - BGP peering and end-to-end IP reachability.

   (a) Based on the defined low-level intents, provide a proof-of-concept to pre-validate network changes in an IBN infrastructure on legacy devices before they are deployed on to the network.

3. **RQ3** Design a method to have a common platform to manage network configuration formats in a vendor-agnostic format.

1.4 Thesis Outline

This thesis report is structured as follows:

- **Chapter 2** introduces the reader to the necessary background concepts involved in this thesis - beginning with introducing Intent-based networking, intents, IBN architecture and the concept of network verification.
- In **Chapter 3**, we define what network validation is and introduce Batfish as a network analysis tool.
- **Chapter 4** discusses the IBN architecture, the design challenges and decisions involved in defining a framework for traditional non-programmable network devices. This is followed by the detailed design of the pre-validation system and the intent database (with two specific types of intents created).
- **Chapter 5** explains how network configurations can be represented in a vendor-agnostic manner, which is a necessary integration with the IBN framework.
- Results from the thesis are discussed in **Chapter 6**, while **Chapter 7** concludes the report along with future scope of work.

Chapter 2

Theoretical Background

In this chapter, the main concepts involved in this thesis will be discussed along with the current state of the art.

2.1 Intent-based networking

As stated in Chapter 1, the basic objective of Intent-based networking is to take the complexity of network management away from the network operator - this includes taking inputs on *what* needs to be done, instead of *how* the objective can be achieved. This ideally involves a central network controller which orchestrates all the necessary steps needed, starting from obtaining the objective from the network operator till the appropriate network commands are applied to the network devices to achieve said objective. The objective that the network operator desires is supplied in the form of an *intent*.

2.1.1 What is an intent?

According to RFC7575 [17], an intent is described as *"An abstract, high-level policy used to operate the network, which does not contain configuration or information for a specific node"*. Thus, an intent is a set of operational goals that a network should satisfy by specifying the outcomes that needs to delivered, without specifying how to achieve them. Intents are defined in a declarative way which describes *what* need to be achieved. Thus, the two main salient features of an intent are [7]:

- Data abstraction: The network operator should not be concerned about the complexity of low-level device configurations.
- Functional abstraction: The network operator should not be concerned about whether a particular outcome is achieved. They are only required to specify what needs to be done.

In order for these intents (high-level policies) to be applied onto the network devices, first they must be broken down into primitive form so that they can be converted into network configurations. These are called *low-level intents*. In an

ideal intent-based networking environment, this process is automated and does not require operator intervention.

The goal of the IBN system is to take away the complexity of device level configurations from the network operator by creating a set of automated actions which will ultimately satisfy the intent decided by the operator. Hence, it is on the IBN system to convert the network operator's intents into machine understandable network configurations to satisfy the objective.

2.1.2 IBN Architecture

IBN is currently a hot topic in the field of network automation, hence there have been multiple works which have attempted to define how an architecture should be defined. There is no convention which needs to be followed for defining an IBN architecture, however there have been works done in the past which give a baseline to build and design the architecture needed for this thesis. Cohen et al. [8] designed an IBN reference architecture using an intent-based North Bound Interface (NBI) and a network overlay abstraction achieved by Distributed Overlay Virtual Ethernet network (DOVE). However, details on the NBI have not been elaborated, nor the actual design of the system implemented.

Cerroni et al. [5] started with an objective of defining an open, vendor-agnostic, and inter-operable NBI and developed an architecture for end-to-end services accross multiple network domains like Internet-of-thingss (IoT), SDN and Cloud. Their architecture was more specialized to the use case considered and thus did not attempt to standardise it for IBN in general. Han et al. [13] designed a layer-based architecture for an intent-based virtualized network. They put forth the concept that each layer in the architecture should serve as an abstraction layer for the layer above, thereby splitting the entire architecture into five component layers: protocol adaptation, abstraction, virtualization, virtual abstraction, and intent layer.

Riftadi et al. [32] [31] inferred a general IBN architecture in their implementation of intent-based networking with P4. Their architecture consisted of intent definition getting converted to network level policies, followed by conversion to service definitions which can be implemented onto programmable network devices. The architecture also involves a monitoring system which ensures the necessary policies are being enforced.

The network industry has also attempted to define IBN architectures. Cisco defined an architecture with a centralized network controller while accomplishing a closed-loop system serving 3 main functions: translation of network intents, deploying these intents to the network and assurance that these intents are being continuosuly enforced on the network [6]. Juniper took a similar approach with their Juniper Apstra IBN software system. This is mainly centered on the design, building, deployment, and operation of data center networks while focussing on zero-touch deployment and continuous validation [20]. Nokia defined

a Network Services Platform (NSP) which also has a centralized network controller and an open programmable platform that enables engineers to automate network operations [26].

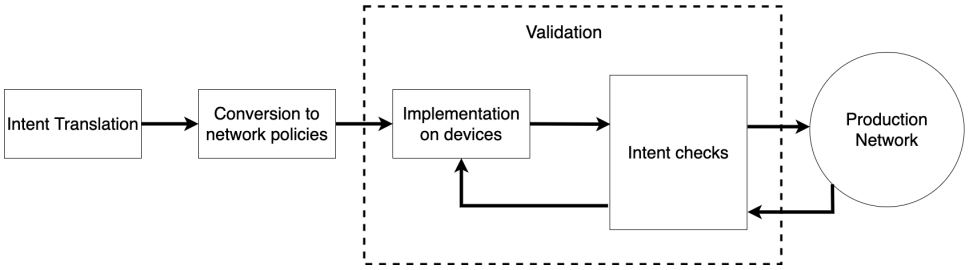


Figure 2.1: Basic IBN Architecture

All of the aforementioned works have their own design strategies and features. However, the problem is that the IBN based infrastructures in these works involve solutions for integrating Software-defined or programmable network devices, i.e., they do not take into consideration the current market share of traditional non-programmable devices and the need for defining an IBN solution for them. The logical approach to addressing traditional devices would be to make dedicated tailor-made solutions per each vendor, but that adds an additional layer of complexity in the system when devices from more than one vendor is present in the networks (which is practically almost every commercial network). This generates a research gap to expand the current state-of-the-art for designing an infrastructure involving multi-vendor traditional non-programmable devices. Using inferences from the above works, a generalised framework for the IBN infrastructure has been designed in this thesis. A basic IBN framework is shown in figure 2.1. A detailed description along with research on design choices is presented in later chapters.

2.1.3 Network configuration representation

For an IBN infrastructure which needs to address network devices from multiple vendors, there is a need for a common network configuration representation format. This is necessary because network devices from different vendors have their own CLI for implementing configuration changes. In this thesis, we take advantage of the fact that the Batfish tool builds a common data plane model for all vendor devices [15]. Since widely used common YANG [16] representations like OpenConfig [27] exist for the purpose of collaboration amongst network operators, there is currently a need for converging a common network configuration format along with an IBN infrastructure to support true multi-vendor functionality. The novel solution addressing this need is explained in detail in chapter 5.

2.2 Network Verification

Network verification and validation (described in detail in chapter 3) help the network operators in asserting whether the changes that have been carried out in the network satisfy their chosen objectives. This is important in an IBN infrastructure as configuration errors also need to be detected and rectified by the network controller, to prevent a compromise in the performance of the network. There are two main approaches that researchers use to analyze network configurations and to detect errors: static analysis and dynamic data-plane analysis.

The static approach involves directly analyzing network configuration files to proactively detect errors before changes are deployed to the network. Configurations of modern network devices have many interacting modules and protocols (like BGP, OSPF, VLANs, ACLs, MPLS, L3VPNs, etc.) and hence, performing a 'what-if' analysis of the configuration files becomes complex. Existing static configuration tools attempt to overcome this complexity by building their own customized models for each feature in the configuration and other properties [1][25][29][36][9]. One such example is 'rcc', the router configuration checker, which detects faults by building a normalized configuration representation [9]. rcc was designed to detect two types of faults - route validity faults and path visibility faults - and verifies a range of errors by checking corresponding properties. FIREMAN is another static analysis tool which specializes in firewall modelling and analysis [36]. By taking advantage of the finite state nature of firewall configurations, it analyzes them by performing symbolic model checking for all possible IP packets and their data paths. It also represents ACLs as specific 'rule-graphs'. The static approach is highly specialized on particular features and relies on customized data models, which makes it limited in scope for what can be checked and consequently requires network operators to look for multiple tools to ensure all network errors have been identified.

The second popular approach to analyzing network configurations is by analyzing the forwarding behavior of the network using data plane snapshots. In contrast to static analysis, data plane analysis detects any unwanted forwarding faults because the data plane is a result of the combination of all aspects involved in the network configurations. Kazemian et al. [22] designed NetPlumber, a Header Space Analysis (HSA) based real time network policy checking tool which maintains a dependency graph between policy rules. They claim that NetPlumber is a natural fit for SDNs, but the underlying model can be used for other devices as well. In another work, Kazemian et al. [23] implemented 'Hassel' a packet header analysis tool which analyses a variety of network protocols. Anteater is another network analysis tool developed by Mai et al. [24] which does static analysis of the data plane. Here, the approach is to convert high-level network invariants into a boolean satisfiability problem (SAT) and compares them using a SAT solver. Zeng et al. [38] also implemented a network verifiying tool called 'Libra' which scales forwarding analysis for larger networks by making use of MapReduce [14]. Data plane analysis is not a proactive approach to detecting network faults, i.e., they cannot detect faults before the undesirable forwarding occurs. This still causes two big challenges

for the network operator. First, even though the faulty forwarding has been identified, the corresponding configuration lines causing the fault still need to be narrowed down manually by the operator. Secondly, an impact of a configuration on forwarding need not be immediate - which means that an erroneous configuration may manifest quite some time after the change has been made.

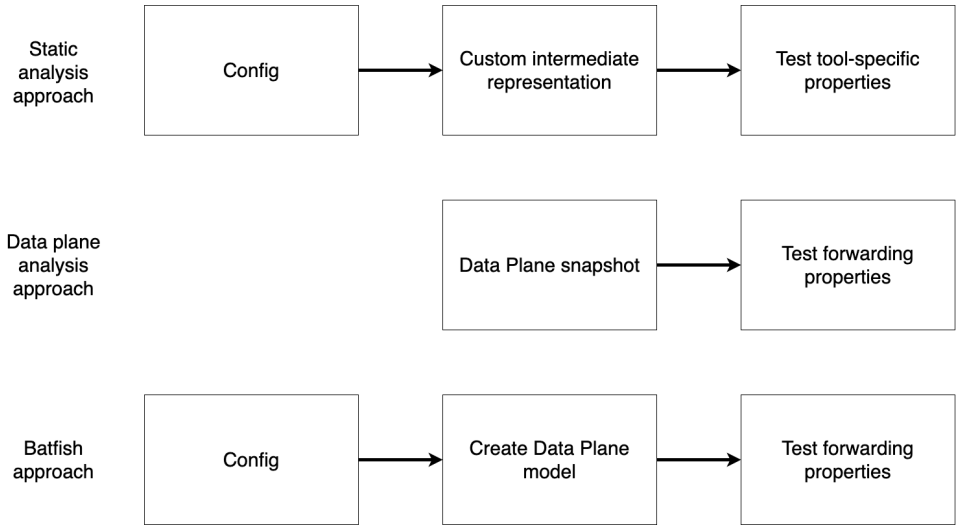


Figure 2.2: Batfish approach [10]

Fogel et al. [10] created the network analysis tool 'Batfish', which combines the advantages of the above two approaches, as illustrated in figure 2.2. Batfish scraps the need for a customized representation, which is a drawback of static configuration analysis methods and also the need for actual forwarding to build a data plane. Instead, it builds a data plane that would eventually be the result of a set of configurations in an environment. This gives the advantage of a proactive approach to detect errors before they are deployed into the network, while also provinding a platform to conduct correctness checks on forwarding properties which can only be inferred from the data plane. 'Minesweeper' is also another tool created by Beckett et al. [2] to translate the protocol information from OSPF, BGP, static routing, etc., present in the configuration files, into a logical formula which best represents the stable state to which the network forwarding will converge.

There are advantages and disadvantages to every network verification tool. In this thesis, Batfish has been chosen as the verification tool to pre-validate network changes in the IBN architecture. The reasoning behind choosing Batfish is explained in more detail in chapter 3.

Chapter 3

Network validation

This chapter describes what network validation is and how it forms an integral part of IBN systems while automating network management. We will also see how networks can be pre-validated and how we choose a specific tool for performing pre-validation, namely, Batfish.

3.1 Defining Network Validation

One of the primary tasks of network engineers and administrators is to configure and make changes on network devices. This usually involves the following steps:

1. Designing and planning a network based on a network requirement
2. Creating network device configurations for that design plan
3. Committing these changes to the network and
4. Ensuring the applied configurations adhere to the design plan and satisfy the initial requirement.

Historically, these tasks were done manually, which were not only time-consuming and laborious, but also highly error-prone since a single mistake could potentially bring down an entire network [3]. To overcome these challenges, network engineers are moving towards automating network tasks which reduces human interaction (and thereby potential of human error). The aforementioned steps also serve as a baseline towards automating changes on the network.

While automation removes the need for physical intervention from the administrator to perform operations (logging into devices, extracting information, pushing configuration changes, etc.) on the network, it does not inherently evaluate the correctness of the operations being carried out. Since all networks need to work according to a design and to satisfy a set of operational requirements, evaluation of the functioning of networks before and after making any changes in the network is of high importance.

The process of validating changes after they are deployed onto the network devices is called **post-validation**. Alternatively, the process of validating changes before deploying them is called **pre-validation** [30]. Both pre-validation and post-validation have their own purpose in network validation.

With post-validation, the functioning of the network is correlated to the requirements after deploying the changes to the network. This means that configurational errors can only be captured by studying the impact it has had on the network through live monitoring or fault management. It answers the question *"Did the change I made just now make the network function in the way I intended it to?"*. By performing post-validation, the network administrator ensures that the impact time is minimised after deploying the changes [30].

In case of pre-validation, the correlation study of the changes made with respect to requirements is done more proactively i.e, before the changes are deployed to the network. This enables the network administrator to ensure that errors don't reach the network before the changes are even deployed, thereby providing a higher degree of protection to the live network as compared to post-validation techniques [30]. Pre-validation gives the answer to the question *"If I make this change in the network, will it still function in the way I intend it to?"*. Although pre-validation may not be able to detect all the errors before changes are deployed (since there may be faults which can only be detected during runtime) it is a crucial step in evaluating the correctness of the changes being planned on the network.

Validation vs verification

When it comes to network automation, there are two terminologies which are used frequently - Verification and Validation. **Validation** is the process of determining whether the overall functionality of the network meets the design requirements and specifications. Validation helps predict how a potential change could affect the network and also to diagnose network behavior which deviates from the norm [30][3]. **Verification**, on the other hand, is the process of determining the correctness of all possible scenarios within a specified *context* - for e.g., ensuring all DNS packets can reach the DNS server. A combination of multiple verification methods may constitute the overall network validation process. Logically, validation is the overall final step in a process, whereas verification is ideally done at intermediate checkpoints [35].

3.2 Batfish

In this section, we will see what Batfish [15] is, and how it can be used for network pre-validation using the capabilties it provides.

3.2.1 What is Batfish?

Batfish is an open-source multi-vendor network analysis tool that allows the user to validate configuration data, query control plane state, verify ACL rule sets, analyze routing/flow paths, as well as simulate network failure [10]. This

it does by creating a 'snapshot' of the network using information such as device configurations, IP Tables, Layer-1 topologies, etc. It is a containerised service which runs offline, which means that it does not need access to online libraries while building its own vendor-agnostic models for network analysis. Once these models are built in the container, they can be queried using the 'pybatfish' Python library to obtain the desired information related to the network.

3.2.2 Batfish as a network pre-validation tool

Batfish has a range of features which help in network validation and analysis which would otherwise be difficult to accomplish in traditional methods, like sending actual packets across the network [10]. Some features which make Batfish ideal for network pre-validation are as follows:

- **Impact analysis** - Helps analyse how the network responds to faults such as link or node failures.
- **Configuration analysis** - Ensures the correctness of the device configurations and whether the network functions according to the configured parameters. In addition, Batfish can also be queried for protocol specific configurations like BGP, OSPF, Interface properties, etc.
- **Packet forwarding analysis** - Batfish has the capability to perform virtual traceroutes and reachability tests which help to assert whether two nodes are connected without resorting to conventional methods like sending actual IP packets. This includes analysis of specific packet header parameters like application (e.g., SNMP) or transport level (TCP, UDP ports).
- **Multi-vendor support** - Batfish functions on vendor-agnostic network models which can be queries by the user to analyse the network. This provides a wide range of support for popular network device vendors like Cisco, Juniper, Arista, Cumulus, etc.

One of the drawbacks of Batfish is the lack of possibility to analyse/simulate real-time packet flows, which might be needed to evaluate throughput and packet loss based requirements. In addition to this, Batfish builds a control plane model using the configuration files provided to it, hence it cannot predict physical network faults which might arise due to hardware related failures. However, the advantages far outweigh the disadvantages when it comes to pre-validation scenarios.

In the following chapter, we will cover the design aspects of the IBN architecture for traditional non-programmable network devices, and where pre-validation plays a role in its design.

Chapter 4

IBN architecture design

In this chapter, we will discuss the design decisions made for the IBN architecture designed in this thesis and where the vendor agnostic pre-validation plays a role in its design. This will aim to address RQ1 of the research questions listed in the introduction chapter.

4.1 Design challenges

In section 2.1.2, the existence of a research gap was underlined to expand the current state-of-the-art for designing an infrastructure involving multi-vendor traditional devices. However, an existing IBN architecture design would not be sufficient in addressing the same. In this thesis, the following challenges were taken into consideration and addressed to design a novel IBN architecture for traditional devices:

- The IBN architecture should be **modular**, i.e., it should be capable of accommodating any type of solution for the individual sub-components. It should also be be easily **scalable** for future work.
- The architecture must be fairly **simple** and straightforward, while at the same time not compromise on the necessary building blocks of an ideal IBN architecture framework.
- It must make sense universally and be **applicable for all types of networks**.
- The design must **not contain vendor-specific components** for realizing the entire IBN system - failing which counter the design objective of building a vendor-agnostic system.

4.2 Defining the architecture

There are many components in a system that help define a truly Intent based provisioned network. A basic skeleton for all the required components in an IBN infrastructure was illustrated in section 2.1.2. Keeping this basic infrastructure and the design challenges listed in section 4.1 in mind, an IBN framework for

the required objective was designed as illustrated in Figure 4.1. This shows three main aspects (sub-divisions) of such a system, and a basic idea how these components will interact. To account for the actual breakdown into functions for the modularity, the sub-divisions - Intent Translation, Validation and Implementation - have been designed to consist of smaller functional blocks. These functional blocks are then interconnected based on the actual flow of the IBN process to form the overall IBN architecture design.

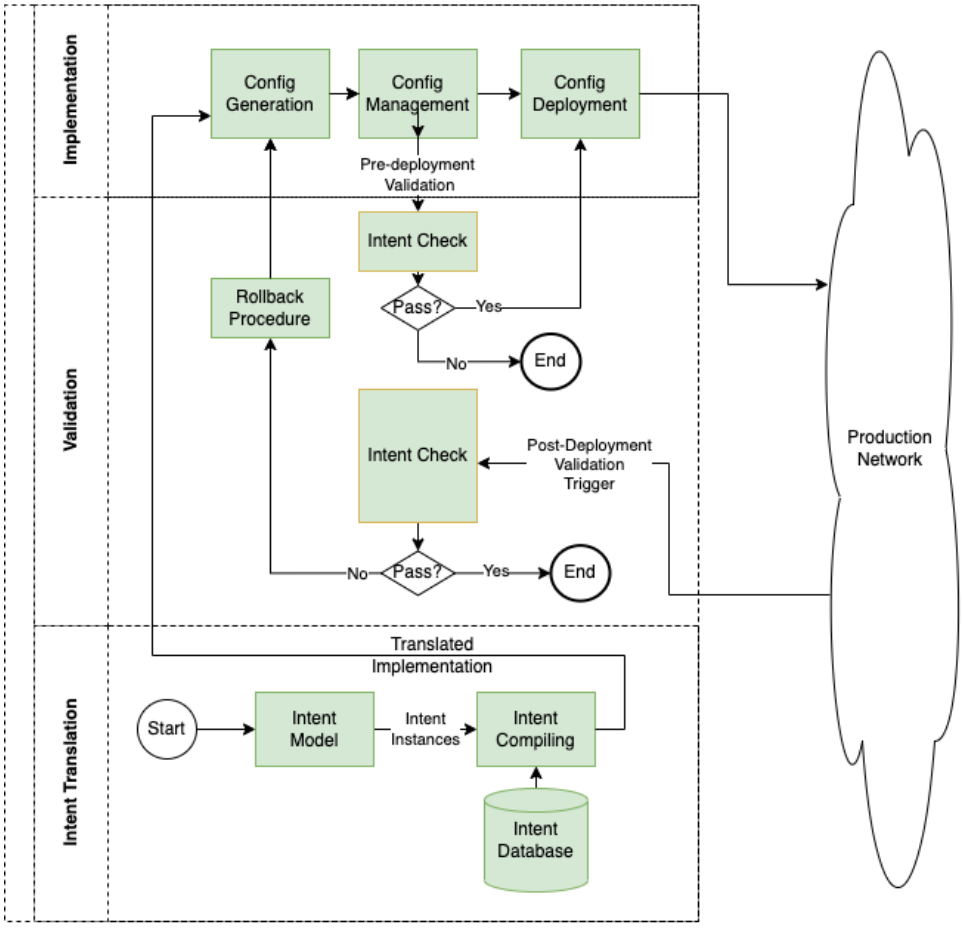


Figure 4.1: Flow in the IBN Architecture

First is the **Intent Translation** sub-division, which covers how the inputs from Network engineers, using these systems, are interpreted and understood in terms of networking needs and system parameters. As per the intent definition stated in section 2.1, a high-level intent in natural language needs to be converted to lower-level intents which can then be used to generate individual network configurations for the devices. The translation of intents is done using Intent Definition Languages (IDL) (e.g., Nile [19]) and stored in the Intent Database.

Intents definitions stored in the Intent database can then be compiled to build network configurations based on the requirements ('Intent Compiling' block). The actual translation process of high-level intents from natural language into low-level intents is not in the scope of this thesis. However, low-level intents are taken into consideration in this thesis (as described later in section 4.3.1) and is used in the demonstration of the proof-of-concept of IBN infrastructure. These low-levels intent structures contain lesser information than what is needed for the actual configuration of the protocol on the network device, thereby still adhering to the simple nature of intents by definition.

In this way, clear instructions about system needs and parameters can be conveyed to the proper provisioning systems as represented in Figure 4.1, between the output of the 'Intent compiling' block and the input of 'Config generation' block. A proof-of-concept of how these low-level intents are gathered and defined is explained in detail in sections 4.3.3 and 4.3.4.

Second is the **Validation** sub-division, once configurations have been interpreted and defined from the intention set, they can be verified and validated in a closed loop via network validation tools like Batfish (see Figure 4.2]). Closed-loop validation is responsible for checking whether the desired intents have been satisfied in the functioning network. As explained earlier, this comprises of both pre- and post-validation mechanisms. For defining a pre-validation system of an IBN architecture designed for traditional devices, the following design challenges exist:

- The core logic of the pre-validation system must **provide the required and appropriate information** to validate the network based on the provided network intents.
- The pre-validation sub-system must be **vendor-agnostic**, i.e., it must be able to validate intents for any type of vendor devices.
- The system must be **scalable** for number of devices and intents, i.e., the processing time of the entire system must increase at a linear scale when number of devices and/or intents are increased.

For addressing these challenges in an IBN system, a proof-of-concept pre-validation pipeline has been designed and tested using Batfish, as shown in Figure 4.2. This is explained in detail in section 4.2.1, while the testing and results are explained in chapter 7.

The third sub-division in the architecture is **Implementation**, where existing systems within the automation framework take care of delivering configuration into the devices in the network. The function of this would be that the required APIs and functions trigger these mechanisms once translation and validation are completed. Since the goal of the overall architecture is also to be vendor-agnostic in nature, it is necessary to manage configurations in a common format. The use of OpenConfig towards achieving this objective is described in chapter 5.

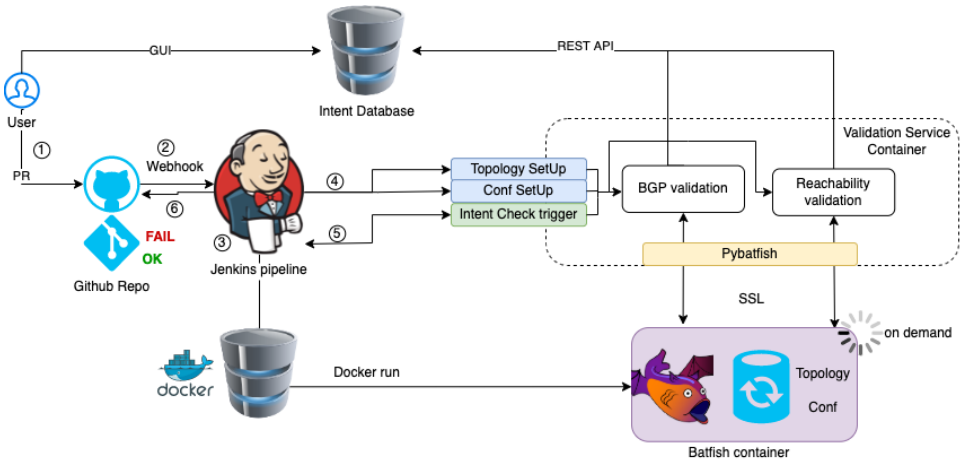


Figure 4.2: The designed Batfish based closed-loop pre-validation system

Figure 4.2 shows the designed approach of a closed-loop network pre-validation implementation with a Docker containerized Batfish service. The system avails itself of a pipeline/workflow execution system, a configuration repository to store the data, the validation framework (i.e., Batfish), the intent database (which stores all network validation information), and the validation element (called SNAP validation which interacts with Batfish).

4.2.1 CI/CD pipeline

Continuous Integration and Delivery (CI/CD) pipelines are integral to automating software processes. They provide several advantages to the operator including earlier defect discovery, higher productivity and process modularity [37]. There are several popular CI/CD tools which are available including Jenkins, Gitlab, Travis CI, Go CD, CircleCI etc., with each having its own advantages and disadvantages. For the purpose of the design in this thesis, we choose Jenkins as the CI/CD tool due to the following reasons [12]:

- It is open source - which means that it is low cost and is being continuously improved by contributors around the globe.
- Variety of features - it has a large number of plugins available to perform necessary actions while also having a friendly user interface
- Scalable - It is highly scalable for automation of large projects.
- Compatibility - Jenkins is compatible with Github and other repositories which makes pipeline automation much easier.

In more detail, the steps depicted in the form of numbered arrows in Figure 4.2, represent the following steps in such pipeline:

A pre-validation system has been designed in this thesis to address the challenges that exist in defining a pre-validation system for traditional devices. As

stated in chapter 3, pre-validation involves evaluating configuration changes before they are deployed on the network devices. For a pre-validation system to work in an IBN system, there is a requirement for:

- Reading/storing network intents, i.e., the intent database,
- An tool which evaluates network protocols in a vendor-agnostic manner (Batfish, in this case)
- A central logical entity which interacts with the database and Batfish to evaluate network intents and send a logical check to the Configuration Management part of the IBN pipeline.
- An overall orchestrator which triggers the above functionalities (in the form of a CI/CD pipeline).

The structure in 4.2 has been designed with the above requirements. Batfish, by itself, is not a pre-validation tool - it merely forms a network snapshot based on which information can be extracted. The real logic of IBN pre-validation is done by the 'Validation Service Container'. For the proof-of-concept, the process works as follows.

Assumption: For the closed-loop validation process, all intents to be validated have to be stored in the Intent Database, as depicted in Figure 4.2. The process how this data can be filled in by the user is explained in detail in section 4.3.3.

1. A change in a version control system (e.g., Git) that publishes a change in network configuration.
2. Changes in the version control system that triggers a change through a CI/CD mechanism (e.g. Jenkins via API Webhook).
3. The CI/CD broker, that given a potential change of configuration of the system, instantiates batfish microservice in the form of a docker container.
4. Once that Batfish instantiation is done, it gets the required inputs (network configurations) through a REST API interface, to perform cross reference checks.
5. The actual trigger of Batfish checks against the intents stored in the Intent Database is done over two independent stages – BGP validation and Reachability validation. Each of these stages is triggered by the CI/CD broker. This is done through a REST API interface.
6. The broker, being able to discern the results of the intent check API call to the system, would be approving/rejecting the changes proposed based on these intentions.

The Batfish interfacing component, mentioned in steps 4 and 5 above, contains services that allow parametric inputs of configuration, topology and intents. Batfish as a tool needs this input to understand the network change or service to be implemented in contrast to the intent. These intents that are to be validated using Batfish, are categorized and stored in a particular manner in the Intent Database, so as to comply with operational requirements (details explained in section 4.3.4).

4.3 The Intent Database

In order to store and manage intents, a database is necessary. Depending upon the requirement of the IBN system, a database can be chosen appropriately to store intents. For the scope of this thesis the Intent Database implementation makes use of VCMDB (Viasat Configuration Management Database) to store and read intents. The reason for choosing the database is explained in the following section.

4.3.1 VCMDB database

Viasat Configuration Management Database (VCMDB) is a 'schema-less' data service, that can store any kind of data, and can be accessed via a web API and a visualization tool. It is technology agnostic and stores an inventory of assets called *resources*, as well as any relationships between these assets. The intelligence of the system is in the data, not the schema itself, which enables dynamic content management without schema changes and the associated downtime. VCMDB also has the ability to paint an end-to-end picture and provide holistic views of an always expanding and evolving ecosystem. Furthermore, it has the capability to establish 'relations' between the database objects, which will prove useful in future works where there is a need to interlink intents between different devices.

The main disadvantage of using VCMDB would be its non-open-source nature and a scope of using it only within the organization. A standard open-source database like mySQL or MongoDB could also be used to store intents, but that would come at the trade-off of the above features. Considering a recommendation from Viasat for using VCMDB database for this proof-of-concept, it is chosen as the Intent database.

Each entry (or VCMDB resource) in the database represents one intent which is unique to the entire network; it also specifies the network parameters that need to be satisfied for that intent. From the VCMDB perspective, each resource in the database is defined by the following key fields:

- **resource_type** – Describes the type of database resource. Under the IBN case, the *'underlay_intent_reachability'* and *'underlay_intent_bgp'* resource type values are used accordingly.
- **name** – Unique user-readable identifier for each intent in the whole database. Under the IBN case, the construction of the name is described in section 4.3.2.
- **config** – This is the field which contains the key network parameters which describe the intent itself. A standardized set of values is used for specifying BGP and reachability intents, which is being specified in the section 4.3.3.

For the proof of concept, two lower level intents have been taken into consideration - BGP peering and IP reachability. This will be explained in detail in section 4.3.3.

4.3.2 The intent nomenclature

To differentiate between Reachability and BGP peering intents, a naming convention has been defined which follows the rules below.

1. For BGP peering intents, the naming of the VCMDB resource in the database is as follows:

   `underlay_intent_bgp-{src_node}-{local_ip}-{remote_ip}`

   where:

   - src_node – Hostname of the originating device
   - local_ip – BGP peer IP in the originating device
   - remote_ip – BGP peer IP in the destination device

2. For reachability intents, the naming of the VCMDB resource in the database is as follows:

   `underlay_intent_reachability-{src_node}-{src_ip}-{dst_ip}`

   where:

   - src_node – Hostname of the originating device
   - src_ip – IP address in the originating device, for which reachability needs to be checked.
   - dst_ip – IP in the destination device, for which reachability needs to be checked.

4.3.3 The 'config' field

As briefly explained at the beginning of section 4.3, the config field contains the network data of the intent, which is necessary to be applied/satisfied on the network. Since BGP and Reachability intents are different in their goals, two different data structures have been defined. Inside the 'config' field, the parameters are stored in JSON formatting.

The BGP intent

The BGP peering intent is to check the session status of a pair of BGP neighbors. Although the BGP protocol, in general, has many more parameters to consider during pre-validation, in this current proof-of-concept the intent involves a binary Established/Not-established status which will be verified using Batfish. As mentioned earlier in section 4.2, the actual translation of high-level intent to lower-level intent is beyond the scope of this thesis. Hence for the purpose of this proof-of-concept, a lower-level intent for BGP peering has been designed as per the structure shown below. This design covers the basic information necessary for validating a BGP peer pair, while also maintaining a level of simplicity which is understandable to network operators. The system shall then be able to validate the session status of such BGP peering through the data contained in this intent.

Batfish validates the BGP sessions by creating a 'snapshot' in the docker container based on the configuration file of the device provided through a GIT repository. Once the snapshot has been created, the python library 'Pybatfish' can be used to ask 'questions' towards the Batfish service to retrieve BGP specific information (this includes 'established' session status).

For creating/modifying a BGP intent, the following structure is in place in the database. It can be seen that the above structure has been designed such that it contains parameters that do not extensively cover the requirements for setting up/evaluating a BGP peering session with a standard CLI based configuration. This enables the intent to be independent of device vendor while also retaining the simplicity characteristics of an intent:

```
{
    "local_ip": "10.37.192.23",
    "src_node": "node_name ",
    "local_asn": "123456",
    "local_vrf": "aabbcc",
    "remote_ip": "10.37.192.22",
    "bgp_status": {
    . . .
    "batfish_status": "<Not-Established/Established>"
    . . .
    },
    "remote_asn": "123456",
    "remote_vrf": "xxyyzz"
}
```

In the structure shown above as an example, fields include:

- **'src_node'** - the network device (its hostname) originating the BGP connection.
- **'local_ip'** and **'remote_ip'**, indicate the BGP peering to be established from the device 'src_node', using IP address 'local_ip' to the neighbor peer whose IP address is 'remote_ip'.
- **'bgp_status'** - indicates the detected BGP session status. Each validation framework is allowed to report its own status inside, in order to allow future extensibility. The allowed values for these entries are Not-Established (for successful verification) or Established (for unsuccessful verification).
- **'local_asn'** and **'remote_asn'** - indicate the values that should be used and validate for the ASN, used locally on the device and remotely on its peer (respectively).
- **'remote_vrf'** and **'local_vrf'** - instances that are used for this connection (also called VRFs), both in the remote device and local device (respectively).

The Reachability intent

This will represent, in the intent database, all 'A-to-B' information that the system should validate in any traffic applicable form (ping, traceroute, TCP and/or UDP). In other words, it represents all devices, and/or sub-instances of them that should be connected to their counterparts needed for a particular circuit or service connection. The current implementation of validation using Batfish is to check the connectivity status between a pair of IP addresses.

A lower-level intent for IP reachability has been designed for this proof-of-concept as per the structure shown below. This design covers the basic information necessary for validating IP reachability, while also maintaining a level of simplicity which is understandable to network operators. Like with BGP in the previous section, Batfish validates the reachability by creating a 'snapshot' in the docker container based on the configuration filed provided. Once the snapshot has been created, the python library 'Pybatfish' can be used to ask 'questions' towards the Batfish service to retrieve reachability specific information based on source and destination IP addresses.

The BGP and reachability intents differ in the way that the BGP intent addresses the protocol messages validation capability of the system during pre-validation, while the reachability intent tests the capability of forming routing tables during pre-validation. This was the reason for choosing these two particular type of intents for the pre-validation proof-of-concept.

```
{
    "dst_ip": "10.37.192.23",
    "src_ip": "10.37.192.22",
    "dst_vrf": "xxyyzz",
    "src_vrf": "aabbcc",
    "dst_node": "dest_node_name",
    "src_node": "node_name",
    "reachability_status": {
    . . .
    "batfish_status": "<Success/Denied>"
    . . .
    }
}
```

In the structure above shown as example, fields include:

- **src_node** and **dst_node** values that indicate the hostnames of the network devices originating and ending a network connection (respectively)
- **src_ip** and **dst_ip** values that indicate the IP values associated from where the traffic should originate to its destination (respectively)
- **src_vrf** and **dst_vrf** values that indicate the routing instances (also called VRFs) where this connection occurs in both the 'src_node' and 'dst_node' (respectively)

- **reachability_status** which indicates the verified connection status. Each validation framework is allowed to report its own status inside, in order to allow future extensions. The allowed values for these entries are “Success” (for successful verification) or “Denied” (for unsuccessful verification).

4.3.4 The Intent database User Interface

To facilitate the usage of intents in a fluid network operation process, a Graphical User Interface (GUI) has been added as part of the IBN system, for creating and visualizing the intents. Since the sole purpose of an IBN architecture is to remove the complexity from the network operators, the main function of the GUI is to provide an easy visualization to all the current intents created on the VCMDB database – BGP and Reachability – and provide a means to create and modify existing ones if needed. While it is not necessary to have a GUI in an IBN architecture, having one is a design choice based on the operator needs.

For the case of Intent creation, when the user (i.e., the Network Engineer) wants to create a new intent, they can do so by the ‘New item’ button option on the GUI panel, wherein all the fields (i.e., network parameters) necessary for creating an entry in VCMDB, can be filled in to create a new resource in the database.

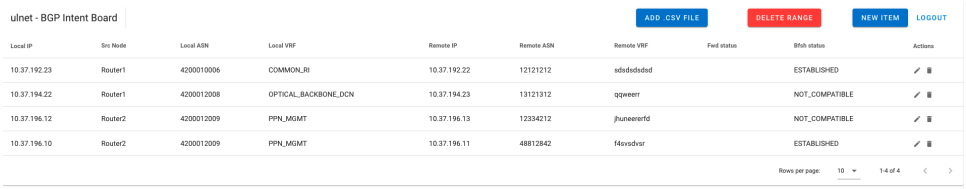


Figure 4.3: IBN BGP intent list visualization from GUI panel

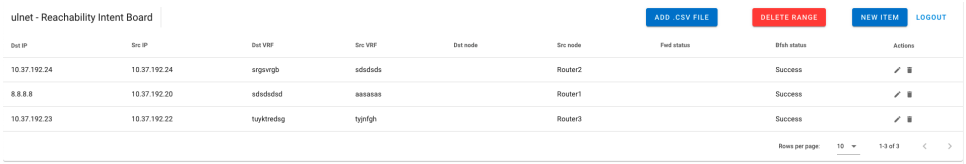


Figure 4.4: IBN Reachability intent list visualization from GUI panel

For the case of intent visualization, the intent data is gathered in the GUI by retrieving existing database resources. Through the backend, API queries for these entries are sent towards the VCMDB IBN API, which are forwarded to the VCMDB database, and then the retrieved information is passed back to the backend. Then the backend uses this information to display it on the GUI front-end as shown in Figure 4.3 for the BGP intents and in Figure 4.4 for the reachability intents.

In summary of this chapter, a new IBN architecture was defined for multi-vendor traditional devices. In the same architecture, a pre-validation system was designed to validate network intents, using Batfish's capabilities. Custom schemas were then designed for low-level intents of BGP peering and IP reachability scenarios.

Chapter 5

Network Configuration Representation

In this chapter we will discuss how network configurations can be represented in a vendor-agnostic manner, which is a necessary integration with the IBN framework.

5.1 Need for a common configuration representation

The vendor-agnostic nature for an IBN system, which serves traditional non-programmable devices, is a primary requirement. As discussed in chapter 4 in the 'Implementation' sub-division of the architecture, one of the main building blocks which addresses this requirement is the 'Config Management' block. This is the module responsible for delivering the configurations to and from the network devices after they have been built from the intents.

In a scenario consisting of multiple vendor devices, it is necessary to have a common platform/structure to represent network configurations. This poses a significant challenge which has not been addressed before as part of an IBN architecture. This thesis overcomes this challenge by providing a design which uses OpenConfig as the middle-ground for device configurations. A proof-of-concept for this is explained in section 5.4. This is done by making use of already existing entity in the system, namely Batfish, and using its data-plane models to convert into OpenConfig.

5.2 Batfish viModel

As mentioned earlier in chapter 2, Batfish creates its own data plane model using the configuration files of network devices. In addition to this, it also supports a wide range of device vendors like Cisco, Juniper, Arista, etc. In order to create a homogeneous data plane to analyse the network interactions between the various types of devices, Batfish creates an internal data model using a common template, called viModel. We take advantage of this to obtain

vendor-agnostic metadata which is converted into OpenConfig YANG models, which will be discussed in section 5.3. The Batfish viModel, in itself, is a custom tree-like structure which contains data parsed from the configuration files. This posed a couple of challenges for justifying and realizing the design path chosen:

- The size of the OpenConfig tree is massive - the full YANG model of all possible protocols and their defined structure in a configuration file is too big to realize all at once.
- The conversion mechanism from vendor specific configuration to OpenConfig needs to be scalable for all protocols.

For this proof-of-concept, the magnitude of addressing these challenges is considerably large and hence, needs to broken down into smaller targets. To begin with, we perform the conversion of interface-related information from two vendor specific configuration files - Cisco and Juniper - into OpenConfig format. This can provide a template for future works on addressing the full conversion of configuration files.

5.3 YANG models and OpenConfig

With a wide variety of vendors for network devices comes their own proprietary CLIs to manage the devices and also to modify and retrieve device configurations. This makes it complex to automate network functions in an environment containing devices from multiple vendors. In 2003, the Internet Engineering Task Force (IETF) designed the Network Configuration (NETCONF) protocol to standardise a network configuration management protocol. NETCONF supported several features which were lacking in protocols such as SNMP [34]. However, this was not enough as a modelling language which was needed to represent the network configurations in an vendor-agnostic format. In order to support this, the YANG modelling language was developed in 2007. The YANG language allows designers to define their own device configuration syntax along with the semantics [34]. Since then, there have been many device configuration representation models which were designed based on YANG.

When the IETF developed YANG, they realized that device vendors needed a data model to model their own versions of YANG. The IETF version of YANG is the simplest version of YANG available to define network configuration models. It uses simpler modules as well as needs fewer lines of code for functioning. Although this is simple to use and implement, this version is very limited in scope for what it can configure between multiple vendors [4]. Hence, adoption of the IETF model in the industry has been slow.

OpenConfig is a model created by a collaborative effort from network operators around the world (not vendors), which is vendor-neutral. The data models of OpenConfig are written in YANG and expands upon the features offered in the native IETF version of YANG. The ultimate goal of the OpenConfig group is to move networks towards a fully programmable architecture and management by having a standardized model of representing device configurations [27].

Having network configurations represented in the format of OpenConfig data models provides a great platform to ensure vendor-agnostic configuration management in an IBN infrastructure. This is the reason why we have chosen OpenConfig as the intermediate representation for all device configurations, irrespective of vendors.

5.4 Combining viModel and OpenConfig

The ideal objective of any IBN framework, including the one designed in this thesis, is to be seamless for network automation involving all types of vendor equipment. This vendor-agnostic framework of IBN requires the system to be independent of device vendors so that network policy definitions are not restricted by vendor proprietary CLI commands.

OpenConfig is an ideal model for representing network configurations and services in a vendor neutral manner. As stated in section 5.3, the OpenConfig YANG model is a collaborative effort from network operators around the world, making it a continuosly improving open-source effort. In this thesis, we take advantage of this vendor-agnostic nature of OpenConfig to define a method which converts vendor-specific configuration files into the OpenConfig format.

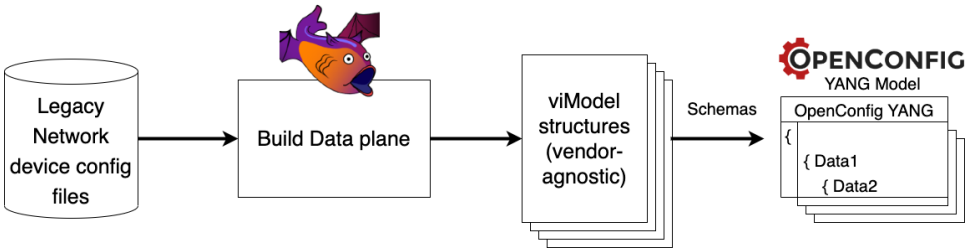


Figure 5.1: Conversion from config files to OpenConfig using Batfish viModel

To realise this, we use the viModel of Batfish, which parses the configuration files into vendor-agnostic metadata, and convert them using schemas into OpenConfig YANG format as illustrated in figure 5.1. As explained before, the scale of realizing all possible protocols in the OpenConfig tree is a long term target and hence, to start with, interface information was extracted from the configuration files of Cisco and Juniper network devices. For the scalability solution, the Marshmallow library of Python was chosen as a template creator for the schemas necessary to convert from Batfish viModel to OpenConfig format. Since the model has been broken down into a schema using Marshmallow, a template method has been set for creating a particular protocol's conversion into OpenConfig format. This means that the methodology of the implementation of the reduced scope in this thesis is applicable for other protocols in the OpenConfig tree as well. Thus, when creating the schemas of future protocol implementations, only the fields needs to be changed.

In the scheme of intent-based networking this provides the following advantages:

1. **Conversion of intents into Services** - The network controller in the IBN infrastructure will need to translate and convert the specified intents into low-level network policies. In case of legacy devices, each device could potentially have a separate vendor proprietary CLI, which makes construction and application of service policies really complex. In case of a vendor-agnostic model like OpenConfig, the network controller can create low-level policies in a common data model which is vendor-neutral.
2. **Comparison of configuration deltas** - Batfish's viModel enables the schematic representation of the current status of the network's control and data plane. This, in combination with the differential comparison of snapshots in Batfish, enables comparison of data planes before and after a config change has been made. When converted to OpenConfig, this can point to where a potential fault might have occured when a change was made.
3. **Abstracting services into configs** - Having a common medium like OpenConfig can also be used to abstract services into configurations. For example, a YANG model has been defined for L3VPN service delivery [18]. Data abstraction at that level can potentially make it easier to convert them into device configurations.

In the scope of this thesis, design has been made for conversion of device information and interface information into OpenConfig format using schemas, in this novel approach.

- A sample conversion from Juniper and Cisco devices has been illustrated in chapter 6.
- As explained earlier, the Marshmallow library of Python was chosen as a template creator for the schemas necessary to convert from Batfish viModel to OpenConfig format.
- The interface information is abstracted from configuration files using Batfish's viModel.
- Since the viModel is a schematic representation of the network's control and dataplane, it can be used to convert the available information into a custom-designed schema made for the OpenConfig format.
- This process of defining the Marshmallow schema of the OpenConfig is based on the OpenConfig protocol tree. Even though the process is time consuming to create a schema for each and every protocol, the advantages it provides for making a vendor-agnostic format for device configurations justifies the means.
- Once the schemas have all been defined in this manner, entire device configurations can be converted to a common, vendor-agnostic model which can be used to evaluate traditional devices in IBN networks.

Expansion of this into all available features from viModel to OpenConfig is work for the future.

Chapter 6

Results

In this chapter we will analyse the results obtained during this thesis. Based on that and, with respect to the research questions set in chapter 1, we will also draw conclusions in chapter 7.

In chapter 4, we illustrated an IBN framework designed for traditional non-programmable network devices. The main challenge of modularity was overcome by defining the architecture into three distinct sub-divisions. Each of these sub-divisions, in turn, needed novel design decisions to address the objectives set at the beginning of this thesis. For these design decisions, the following proofs-of-concept were successfully made:

- Low-level intent definition design in the 'Intent Translation' sub-division (BGP and reachability).
- Network pre-validation pipeline for the above defined intents in the 'Validation' sub-division (using Batfish).
- Defining a vendor-agnostic configuration management method in the 'Implementation' sub-division (using OpenConfig and Batfish viModel).

This involved the full design of the pre-validation module - including Batfish, the design of the intent database and its schemas and also the CI/CD pipeline for the automation of processes.

The test setup

- The CI/CD pipeline automation, including the database, was hosted on Viasat servers.
- To mimic network operators use-case, the pre-validation part of the Proof-of-concept was also tested in a standard computer (16GB RAM, 2.4GHz quad-core CPU).

6.0.1 System performance of network pre-validation

To study the latency performance of the pre-validation system, 'time elapsed' was taken as a parameter of measuring the overall process execution time. This

would present an estimate on the scalability of the system with respect to number of devices in the network and the number of intents. The following points illustrate the process performed to measure the performance and the inferences from the findings:

- To perform the study, 100 different device configuration files were taken (a combination of Cisco and Juniper). The number of network intents was also taken as 100 in the VCMDB database.
- To study the total time taken for the pre-validation process, the number of network device configurations in the network was steadily increased from 5 to 100.
- The overall process was split into 4 chronological phases: Setup time, Batfish snapshot creation time, Intent processing time and Teardown time.
- Time was measured using in-script timer in Python.
- For each successive iteration for measuring latency performance, 5 more device configurations were added.
- For the overall process, the Jenkins pipeline setup time was on average 63 seconds (± 2 seconds). This includes connecting to Jenkins server, deploying 2 Docker containers based on required libraries (one for the Validation service, one for Batfish), loading the scripts and configuration files from Github. This is shown as 'Jenkins setup time' in figure ¡xxxxx¿.
- Once setup, the load time for the scripts and initialization time of the code on average was 0.075 seconds. This is shown in figure 6.2.
- The snapshot creation time was the deciding point of interest for latency. It was noticed that the snapshot creation time increased linearly with number of configuration files while allowing a confidence interval of 5% (as shown in figure 6.2). This means that the system is scalable for higher network sizes, i.e., an exponential rise in delay is not observed, which would prove detrimental for practical use.
- On average, a 2 second time delay was observed for retrieving intents from the VCMDB database. Adding this to the intent comparison time (time for matching the intents with the Batfish results), we get a total average intent processing time of 2.367 seconds (± 0.1522 seconds).
- Once the process was completed, the average teardown time of the Batfish snapshot and the Batfish network was 0.0091 seconds (± 0.0023 seconds).
- The total time elapsed chart as a function of 'Number of configuration files' is as shown in figure 6.2.
- Thus, it can be inferred that the designed system is scalable for practical use according to the latency study of 100 network devices. From the linear trend of the snapshot creation time with respect to number of devices (figure 6.1), it can also be said that the system can scale well for even bigger networks without causing high latency. Hence, the design choices are justified for practical use of the designed system, when it comes to latency performance requirements.

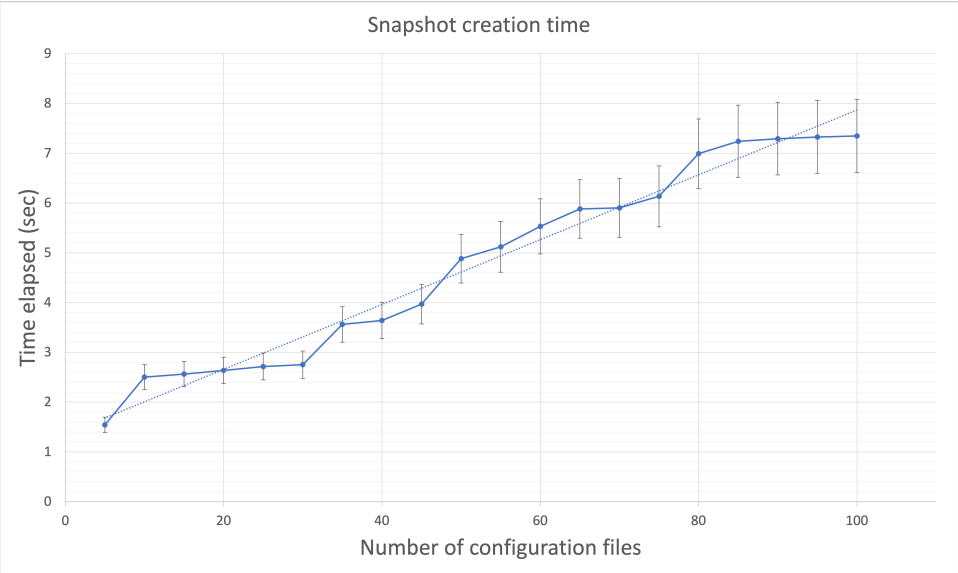


Figure 6.1: Snapshot creation time as a function of number of device configurations

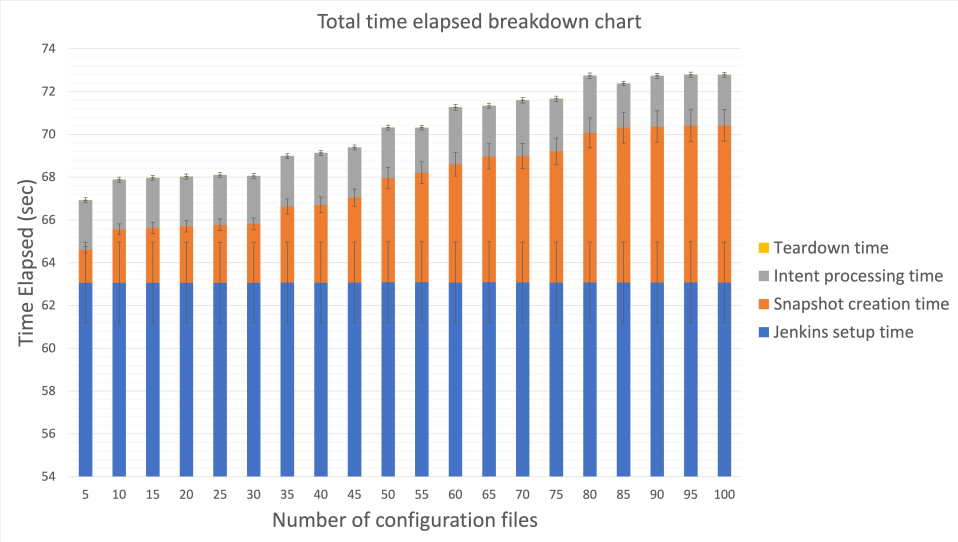


Figure 6.2: Overall time elapsed as a function of number of device configurations

- In addition to this, to test the speed of snapshot switching, approximately 6900 different snapshots of 3 routers each were used as an input. The loading and deleting of all of them together took around 180 minutes.

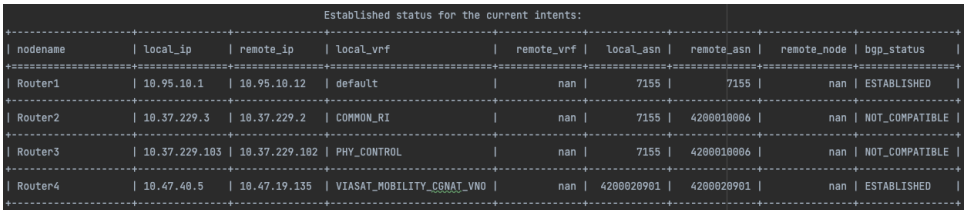
- From the result above, it can be concluded that the snapshot creation time by Batfish is the major latency inducing factor involved in the pre-validation method.

- Overall, the system operates at a practically low latency for pre-validation scenarios. This means that an network administrator can pre-validate a network of approximately 100 devices in around one and half minutes. For a network evaluation on an everyday computer, this is a good result - thereby justifying the design decisions taken.

6.0.2 Network Pre-validation

The system was tested and validated using intents defined for Viasat internal network devices (which have been renamed here) and the following results were obtained:

- **BGP Pre-validation** - BGP peering low-level intents were successfully created in the VCMDB database using the designed schema and the customised GUI. These intents were then pre-validated in the system (Viasat servers and laptop both) to test whether the BGP sessions' statuses were successful or not based on the actual device configurations. Figure 6.3 is an example of a summary of the pre-validation stage implemented in the IBN framework. Each row denotes a separate intent that needs to be satisfied in the system and the 'bgp_status' column signifies the BGP session establishment check done by the system (whether it is ESTABLISHED, DENIED or NOT-COMPATIBLE).



Established status for the current intents:

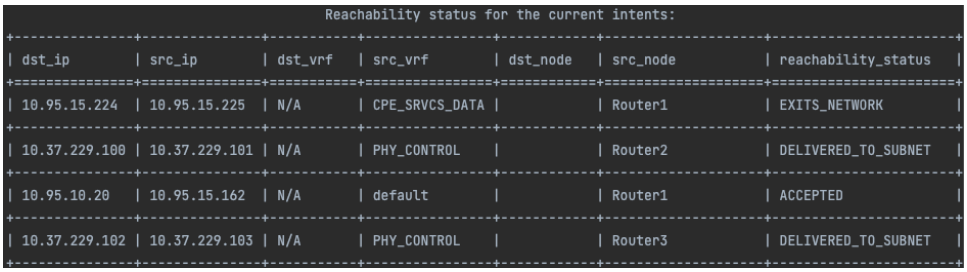
nodename	local_ip	remote_ip	local_vrf	remote_vrf	local_asn	remote_asn	remote_node	bgp_status
Router1	10.95.10.1	10.95.10.12	default	nan	7155	7155	nan	ESTABLISHED
Router2	10.37.229.3	10.37.229.2	COMMON_RI	nan	7155	4200010006	nan	NOT_COMPATIBLE
Router3	10.37.229.103	10.37.229.102	PHY_CONTROL	nan	7155	4200010006	nan	NOT_COMPATIBLE
Router4	10.47.40.5	10.47.19.135	VIASAT_MOBILITY_CGNAT_VN0	nan	4200020901	4200020901	nan	ESTABLISHED

Figure 6.3: BGP Intents status

- **Reachability Pre-validation** - Similar to BGP peering, IP reachability intents were also successfully created in the VCMDB database using the designed schema and the customised GUI. These intents were then pre-validated in the system to test whether the IP reachability from source to destination were successful or not based on the actual device configurations. Figure 6.4 is an example of a summary of the pre-validation stage implemented in the IBN framework. Each row denotes a separate intent that needs to be satisfied in the system and the 'reachability_status' column signifies the IP reachability status done by the system.

The status of the intents were also successfully updated in the VCMDB intent database after pre-validation.

The intent structures created were low-level, i.e., they maintain an understandable simplicity for the user while at the same time contain the basic information necessary for validating the addressed protocol/feature and not too much information so that it moves away from the concept of IBN. From this,



Reachability status for the current intents:

dst_ip	src_ip	dst_vrf	src_vrf	dst_node	src_node	reachability_status
10.95.15.224	10.95.15.225	N/A	CPE_SRVCS_DATA		Router1	EXITS_NETWORK
10.37.229.100	10.37.229.101	N/A	PHY_CONTROL		Router2	DELIVERED_TO_SUBNET
10.95.10.20	10.95.15.162	N/A	default		Router1	ACCEPTED
10.37.229.102	10.37.229.103	N/A	PHY_CONTROL		Router3	DELIVERED_TO_SUBNET

Figure 6.4: Reachability Intents status

it can be inferred that the pre-validation proof-of-concept was successful based on the provided intents and the design choice is a good one.

6.0.3 Network configuration representation

As described in chapter 5, in order to have a vendor-agnostic format for configuration files, OpenConfig was chosen as the data model for representation.

- **Conversion of Cisco config into OpenConfig format** - This is done by first converting the Cisco configuration file into viModel using Batfish (as shown in figure 6.5) and then to an OpenConfig JSON as shown in figure 6.6.
- **Conversion of Juniper config into OpenConfig format** - This is done by first converting the Juniper configuration file into viModel using Batfish (as shown in figure 6.7) and then to an OpenConfig JSON as shown in figure 6.8.

As can be seen from the results, the viModel of Batfish converts the data plane of Cisco and Juniper into a common format. From here, using schemas in Python, these were converted to OpenConfig format according to their corresponding configurations.

The requirement of a vendor-agnostic is re-addressed using this feature. By not only providing a common platform in OpenConfig for representing network configurations, this proof of concept also lays the foundations for expansion into all available features from viModel to OpenConfig.

Figure 6.5: Cisco viModel

```
{
  "interfaces": [
    {
      "name": "FortyGigE0/0/0/30",
      "enabled": false,
      "mtu": 1500
    },
    {
      "description": "MANAGED | routerx | et-0/0/0 | |",
      "name": "HundredGigE0/0/0/0",
      "enabled": true,
      "mtu": 9192
    },
    {
      "description": "MANAGED | routery | 0/0/0/0 | |",
      "name": "HundredGigE0/0/0/1",
      "enabled": true,
      "mtu": 9192
    },
    {
      "name": "HundredGigE0/0/0/2",
      "enabled": false,
      "mtu": 1500
    },
    {
      "description": "\"MANAGED | routera | HundredGigabit0/0/0/13\"",
      "name": "HundredGigE0/0/0/3",
      "enabled": true,
      "mtu": 9192,
      "subinterfaces": [
        {
          "description": "\"MANAGED | router1 | HundredGigabit0/0/0/13.601 |  | COMMON_RI_BACKHAUL\"",
          "name": "HundredGigE0/0/0/3.601",
          "index": 601,
          "enabled": true
        },
        {
          "description": "\"MANAGED | router1 | HundredGigabit0/0/0/13.602 |   | PPN_TO_CN_BACKHAUL\"",
          "name": "HundredGigE0/0/0/3.602",
          "index": 602,
          "enabled": true
        },
        {
          "description": "\"MANAGED | router1 | HundredGigabit0/0/0/13.603 |  | PPN_TO_PPN_DATA_BACKHAUL\"",
          "name": "HundredGigE0/0/0/3.603",
          "index": 603,
          "enabled": true
        },
    {
      "name": "PTP0/RP0/CPU0/0",
      "enabled": false,
      "mtu": 1500
    },
    {
      "name": "PTP0/RP1/CPU0/0",
      "enabled": false,
      "mtu": 1500
    }
  ],
  "name": "router1",
  "system": {
    "dns": {
      "servers": [
        "10.43.0.12",
        "10.43.0.44",
        "10.43.1.12",
        "10.43.1.44"
      ]
    },
    "hostname": "router1",
    "domain-name": "gi-nw.viasat.io",
    "ntp": {
      "servers": [
        {
          "address": "10.137.188.93"
        }
      ]
    },
    "vendor-model": "CISCO_IOS_XR"
  }
}
```


Figure 6.6: Cisco Openconfig JSON

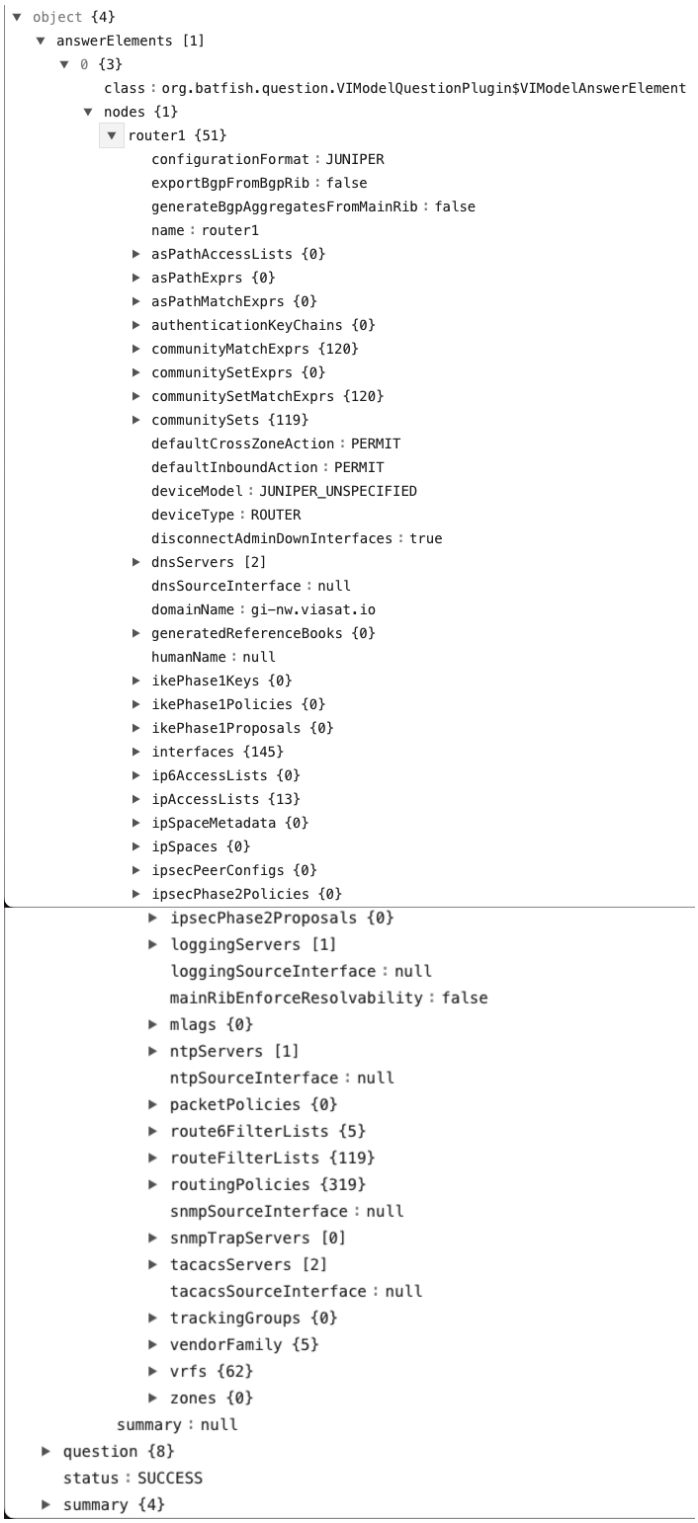


Figure 6.7: Juniper viModel

```
{
  "system": {
    "hostname": "router1",
    "dns": {
      "servers": [
        "10.43.0.12",
        "10.43.1.12"
      ]
    },
    "ntp": {
      "servers": [
        {
          "address": "10.137.188.204"
        }
      ]
    },
    "vendor-model": "JUNIPER",
    "domain-name": "gi-nw.viasat.io"
  },
  "name": "router1",
  "interfaces": [
    {
      "name": "ae13",
      "description": "netsw01/02-vprod | Po-13-14",
      "mtu": 9192,
      "enabled": true,
      "subinterfaces": [
        {
          "name": "ae13.1300",
          "index": 1300,
          "description": "router2 | ae13.1300",
          "enabled": true
        },
        {
          "name": "ae13.1303",
          "index": 1303,
          "description": "router2 | IRB.1303",
          "enabled": true
        },
        {
          "name": "ae13.1499",
          "index": 1499,
          "description": "router2 | ae13.1499",
          "enabled": true
        }
      ]
    },
    {
      "name": "irb",
      "subinterfaces": [
        {
          "name": "irb.2001",
          "index": 2001,
          "description": "SMAC_CTRL Tenant",
          "enabled": true
        },
        {
          "name": "irb.2002",
          "index": 2002,
          "description": "SMAC_DATA Tenant",
          "enabled": true
        },
        {
          "index": 2003,
          "name": "irb.2003",
          "description": "VWA_SAT_DATA Tenant",
          "enabled": true
        }
      ]
    },
    {
      "name": "lo0",
      "mtu": 1500,
      "enabled": true,
      "subinterfaces": [
        {
          "name": "lo0.1006",
          "index": 1006,
          "description": "Internet | Loopback",
          "enabled": true
        },
        {
          "name": "lo0.1010",
          "index": 1010,
          "description": "MGMT_BACKHAUL_TYPE1_HUB | Loopback",
          "enabled": true
        },
        {
          "name": "lo0.10006",
          "index": 10006,
          "description": "FIREWALL_INTERNET | Loopback",
          "enabled": true
        }
      ]
    }
  ]
}
```


Figure 6.8: Juniper Openconfig JSON

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This section concludes the work done in this thesis. This is mainly centered around answering the research questions from chapter 1, based on the findings and results that were obtained during this thesis.

1. **RQ1 - How to design a framework which helps adapt the IBN concept to traditional, non-programmable network devices. This framework should be seamless for all types of network devices and be vendor-agnostic.** Over the course of this thesis, a customized framework was created for the IBN concept to support traditional non-programmable devices. This was illustrated in chapter 4. The architecture is modular, simple and is compatible universally with all types of networks. The modular nature of the architecture was emphasised by the three sub-divisions - Intent Translation, Validation and Implementation. Thus, the IBN concept has been extended to legacy network devices as well.

2. **RQ2 - How to design an intent structure for low-level intents which can be used to pre-validate two popular types of network requirements namely - BGP peering and end-to-end IP reachability.** Section 4.3.3 illustrates the custom schema format created for BGP peering and IP reachability intents. This custom schemas enable network operators to specify intents at a moderately high level in a vendor-agnostic manner without resorting to specifying every parameter that needs to be checked. The intents were stored and managed using a carefully chosen intent database in the form of VCMDB (Viasat proprietary), which was then used to pre-validate on the network.

    - **RQ2(a) - Based on the defined low-level intents, how to pre-validate network changes in an IBN infrastructure on legacy devices before they are deployed onto the network?** After researching multiple pre-validation tools, Batfish was chosen as the best tool for pre-validating legacy device configurations. This is

mainly because Batfish builds a data plane model which is vendor-agnostic and simulates the forwarding scenario based on the current network snapshot. Taking advantage of this, BGP and reachability status of the above mentioned intents can be validated. The design choices in the sub-module of pre-validation have been explained in detail in chapter 4. Batfish also provides viModel, which proved crucial in RQ1 stated above.

3. **RQ3 Design a method to have a common platform to manage network configuration formats in a vendor-agnostic format**.For making the architecture truly vendor-agnostic, OpenConfig was adopted as the means to achieve this goal. Using viModel and schemas created in Python, Proof-of-concept OpenConfig data models were created for Juniper and Cisco devices for interface and node information as explained in chapter 5.

7.2 Future work

For future work, the following areas could potentially provide new areas of research:

- Expand the current architecture to include Intent Definition Languages, and use them to directly convert configurations to OpenConfig format.
- Take advantage of the modular nature of this architecture to integrate with already existing Software-defined IBN frameworks.
- Explore the usage and integration of Machine Learning methods to predict network faults based on network pre-validation data.

Bibliography

[1] E.S. Al-Shaer and H.H. Hamed. Discovery of policy anomalies in distributed firewalls. In *IEEE INFOCOM 2004*, volume 4, pages 2605–2616 vol.4, 2004.

[2] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, pages 75–88, 1984.

[4] CBT Nuggets. Native YANG models: IETF vs Openconfig vs Cisco. `https://www.cbtnuggets.com/blog/technology/networking/native-yang-models-ietf-vs-openconfig-vs-cisco`, Sep 2021. Last accessed: May. 16, 2022.

[5] Walter Cerroni, Chiara Buratti, Simone Cerboni, Gianluca Davoli, Chiara Contoli, Francesco Foresta, Franco Callegati, and Roberto Verdone. Intent-based management and orchestration of heterogeneous openflow/IoT SDN domains. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–9, 2017.

[6] Cisco Systems. Cisco intent-based networking (IBN). `https://www.cisco.com/c/en/us/solutions/intent-based-networking.html`, 2021. Last accessed: May. 14, 2022.

[7] Alexander Clemm, Laurent Ciavaglia, Lisandro Zambenedetti Granville, and Jeff Tantsura. Intent-Based Networking - Concepts and Definitions. *IETF Network Working Group*, Mar 2020.

[8] Rami Cohen, Katherine Barabash, Benny Rochwerger, Liran Schour, Daniel Crisan, Robert Birke, Cyriel Minkenberg, Mitchell Gusat, Renato Recio, and Vinit Jain. An intent-based approach for network virtualization. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 42–50, 2013.

[9] Nick Feamster and Hari Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design Implementation - Volume 2*, NSDI'05, page 43–56, USA, 2005. USENIX Association.

[10] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, May 2015. USENIX Association.

[11] Vikas Gaikwad and Rachita Rake. Software defined networking Market by Component - Forecast 2020–2027. *Allied Market Research*, "Sep" 2020.

[12] Geekflare. 11 Best Continuous Integration(CI) Tools in 2022. `https://geekflare.com/best-ci-tools/`, Mar 2022. Last accessed: May. 15, 2022.

[13] Yoonseon Han, Jian Li, Doan B. Hoang, Jae-Hyoung Yoo, and James Won-Ki Hong. An intent-based network virtualization platform for SDN. *2016 12th International Conference on Network and Service Management (CNSM)*, pages 353–358, 2016.

[14] IBM Corporation. Apache MapReduce. `https://www.ibm.com/topics/mapreduce`, 2021. Last accessed: May. 15, 2022.

[15] Intentionet. Batfish. `https://www.batfish.org/`, 2021. Last accessed: May. 15, 2022.

[16] Internet Engineering Task Force (IETF). RFC 6020 - YANG - A Data Modeling Language for the Network Configuration Protocol. `https://datatracker.ietf.org/doc/html/rfc6020`, Oct 2010. Last accessed: May. 15, 2022.

[17] Internet Engineering Task Force (IETF). RFC 7575 - Autonomic Networking: Definitions and Design Goals. `https://datatracker.ietf.org/doc/html/rfc7575`, Jun 2015. Last accessed: May. 15, 2022.

[18] Internet Engineering Task Force (IETF). RFC 8049 - YANG Data Model for L3VPN Service Delivery. `https://datatracker.ietf.org/doc/html/rfc8049`, Feb 2017. Last accessed: May. 15, 2022.

[19] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining Network Intents for Self-Driving Networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 15–21, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Juniper Networks. Juniper Apstra IBN solution. `https://www.juniper.net/us/en/products/network-automation/apstra.html`, 2021. Last accessed: May. 14, 2022.

[21] Justina Alexandra Sava. Software-defined networking market revenue worldwide 2020-2027. `https://www.statista.com/statistics/468636/global-sdn-market-size/`, Feb 2022. Last accessed: May. 15, 2022.

[22] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *10th USENIX Symposium on Networked Systems*

Design and Implementation (NSDI 13), pages 99–111, Lombard, IL, April 2013. USENIX Association.

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

[24] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 290–301, New York, NY, USA, 2011. Association for Computing Machinery.

[25] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, page 1–8, USA, 2010. USENIX Association.

[26] Nokia Corporation. Network Services Platform. `https://www.nokia.com/networks/products/network-services-platform/`, 2021. Last accessed: May. 14, 2022.

[27] OpenConfig group. OpenConfig. `https://www.openconfig.net/`, 2021. Last accessed: May. 15, 2022.

[28] Lei Pang, Chungang Yang, Danyang Chen, Yanbo Song, and Mohsen Guizani. A survey on intent-driven networks. *IEEE Access*, 8:22862–22873, 2020.

[29] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE Network*, 19(6):12–19, 2005.

[30] Ratul Mahajan, Intentionet. The what, when, and how of network validation. `https://www.intentionet.com/blog/the-what-when-and-how-of-network-validation/`, 2019. Last accessed: Mar. 17, 2022.

[31] Mohammad Riftadi. Intent-based networking with programmable data planes. Master thesis, Delft University of Technology, Delft, The Netherlands, 2019.

[32] Mohammad Riftadi and Fernando Kuipers. P4I/O: Intent-Based Networking with P4. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 438–443, 2019.

[33] Barun Kumar Saha, Deepaknath Tandur, Luca Haab, and Lukasz Podleski. Intent-Based Networks: An Industrial Perspective. In *Proceedings of the 1st International Workshop on Future Industrial Communication Networks*, FICN '18, page 35–40, New York, NY, USA, 2018. Association for Computing Machinery.

[34] Jürgen Schönwälder, Martin Björklund, and Phil Shafer. Network configuration management using NETCONF and YANG. *IEEE Communications Magazine*, 48(9):166–173, 2010.

[35] STC Admin - Software testing Class. Difference between Verification and Validation. http://www.softwaretestingclass.com/difference-between-verification-and-validation/, Aug 2013. Last accessed: May. 15, 2022.

[36] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and P. Mohapatra. FIREMAN: a toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 15 pp.–213, 2006.

[37] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 471–482, 2021.

[38] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 87–99, Seattle, WA, April 2014. USENIX Association.