High-Quality Network Specifications via Coverage Analysis and Relational Abstractions

Xieyang Xu

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2024

Reading Committee:

Ratul Mahajan, Chair

Arvind Krishnamurthy, Chair

David P. Walker

Program Authorized to Offer Degree:

Computer Science & Engineering

University of Washington

Abstract

High-Quality Network Specifications via Coverage Analysis and Relational Abstractions

Xieyang Xu

Co-Chairs of the Supervisory Committee:
Associate Professor Ratul Mahajan
Computer Science & Engineering

Professor Arvind Krishnamurthy
Computer Science & Engineering

Network misconfigurations frequently lead to outages that can impose severe social and economic consequences. Preventing these misconfigurations via automated testing or verification requires complete and precise specification of expected network behaviors. However, creating such specifications is challenging for network engineers due to the enormous scale and rapid evolution of modern networks. This dissertation develops two approaches to address this challenge. The first approach quantifies which network components are covered vs uncovered (and potentially untested) relative to given specifications. Results of this assessment help network engineers refine the specifications to make them more comprehensive. The second approach enables compact specification of network changes, which are the primary source of outages. These specifications capture the similarities and differences between two network states, which makes them complete and precise but compact. Network engineers need only specify the changing parts of the network, which tend to be small, and then simply mandate that all other behaviors "stay the same" without needing to enumerate them. Both approaches have been deployed and shown to be effective in real-world production networks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am grateful of many people for their support and companion throughout the graduate school. This journey would not have been so smooth and joyful without the guidance, patience and inspiration from my advisors, Ratul Mahajan and Arvind Krishnamurthy. I am fortunate to have them as two endless sources of support for any kind of challenge. Ratul masters the education of an independent researcher despite just getting started as a faculty member. His secret source is to start with hands on and gradually let go—and this worked out for me. I am truly indebted to Ratul for being guided step by step in formatting, executing and presenting our first research work. And it is only more beneficial when he intentionally reduced the help to make things more challenging (but not too challenging) for me, while continuing to give insightful and constructive feedback whenever I needed it. Arvind has always been generous for his time chatting with me, even when I was just too stressed to work and he had a heavily booked calendar. I was cheered up almost every time talking about things happening in work and life with Arvind. I am also greatly thankful to Ryan Beckett and David P. Walker, who I consider as two additional advisors of mine. I spent one of my most productive summers working (virtually) with Ryan at Microsoft. He not only introduced me into a promising research problem but also was able to unblock me with just 15 minutes each time when problems were encountered. The research work started during that summer eventually leads to this dissertation. Dave has significantly uplifted the research enthusiasm and quality during each piece of work in this dissertation, with his humour and academic rigorousness. I like to thank my other collaborators, especially Karthick Jayaraman, Weixin Deng, Yifei Yuan and Ennan Zhai. I have enjoyed working with them and learnt invaluable skills and insights from

each of them. I also thank Rene Just and Radha Poovendran for serving on my supervisory committee. This dissertation would not have been possible without Chenren Xu, my undergraduate advisor who introduced me into computer networks. I thank all my friends in Seattle who spent cherishable time with me. Finally, I like to express my sincere gratitude to my parents, my sister, and my dear girlfriend, Yuhao Wan. My journey with your companion starts here and will go long. This dissertation is dedicated to you.

Chapter 1

INTRODUCTION

In this digital age, the Internet serves as a vital infrastructure that underpins countless aspects of daily life. From communication and commerce to education and entertainment, it has transformed how we live, work, and interact, making it an indispensable tool for individuals, businesses, and governments.

Despite its pervasive presence and importance, the Internet is plagued by frequent outages, which capture headlines with alarming regularity. These outages have led to severe and far-reaching consequences across communities and industries, e.g., when planes were grounded (United Airlines 2017), bank accounts went offline (BB&T 2018), emergency calls were rendered inaccessible (T-Mobile 2020), and apps went down for billions of users (Facebook 2021) [67, 57, 76, 58].

A significant portion of these outages stem from mistakes made during the routine management of network configurations. These are commands and parameters that dictate how individual routers operate and interact with other routers to determine how data packets flow in a network. Writing and maintaining correct network configurations requires considerable expertise and attention to the intricate interplay of different network components, making it a notoriously difficult task for humans to execute flawlessly. As a result, Internet reliability remains vulnerable to the fallibility of human operators, motivating the need for effective solutions to identify and repair misconfigurations before they affect the networks.

Over the past decade, many testing and verification tools have been developed. These tools can check whether a snapshot of the network is compliant with a set of user-defined *specifications* that describe desired network behavior. Since the early 2010s, network testing

and verification tools have rapidly evolved from research ideas to production deployments at all large cloud providers [42, 71, 80, 82, 8].

However, network outages caused by misconfigurations continue to occur despite the use of testing and verification methods; we can identify consecutive incident reports from cloud providers despite standard practices of testing and verification [16, 58, 63]. We maintain that ongoing problems are caused by the fundamental challenges of writing and maintaining complete and precise specifications for networks, a critical foundation for testing and verification to succeed.

Two primary factors contribute to these challenges: *scale* and *incomplete information*. As networks continue to expand in size and complexity, the specification needed to describe their behavior grows proportionally. For instance, in our experiments, we encountered networks comprising thousands of routers and millions of configuration lines in total that must correctly forward 10^6 classes of traffic with distinct paths. Creating a detailed specification for these poses an insurmountable task. It is even more complex to update existing specifications after the network evolves because engineers are unlikely to maintain the older specifications that do or do not reflect the updated network.

Adding to these challenges is incomplete information. Networks are built incrementally over years, and their size and complexity necessitate the management of different segments by separate teams. Consequently, individual engineers may possess only partial knowledge of the network's behavior and configuration. In such scenarios, creating an accurate global specification becomes exceedingly challenging since it requires synthesizing insights from multiple sources and precisely reasoning about the interplay between various components, including ones that people may lack adequate knowledge of.

Despite the surge of testing and verification tools, scant attention is being paid to ways to assist engineers in writing more complete and useful specifications. Given the challenges described above and the tight timeframes engineers have to complete testing and verification procedures, the specifications used in existing practice often fall short. They are often *incomplete*, leaving significant portions of the network untested, and/or *overly re-*

strictive, imposing blanket requirements on all network devices without considering corner cases or network heterogeneity. The consequence is an overwhelming number of false alarms and difficulty distinguishing true errors from benign deviations in the results. Thus, the absence or poor quality of specifications significantly constrains the effectiveness of testing and verification efforts and ultimately undermines the reliability of the network infrastructure.

This dissertation addresses the challenges of high-quality network specifications with two novel approaches. The first approach is *coverage-guided specification*. Instead of relying on humans to ensure the completeness of network specifications, this approach *analyzes aspects of the network that have or have not been covered by existing specifications*. Provided this help, engineers can systematically identify the aspects where additional specifications must be added for completeness. The second approach is *relational specification*. It aims to simplify the specifications for network changes, which are the primary cause of network misconfigurations. Unlike traditional specification, which addresses the desired behavior of a single network snapshot, relational specification models two network snapshots (*i.e.*, pre- and post-change) and captures their similarities and differences. They are therefore compact and precise, allowing engineers to simply mandate that other network behavior "stay the same," without enumerating them. Changing network components are usually small and known to the engineers making the changes. Both approaches have been deployed and are shown to be effective in real-world, production networks.

The remainder of this introduction is organized as follows. In §1.1 I highlight the difficulties of writing correct network configurations and the consequences of a failure to do so. §1.2 introduces the key role that specifications play in identifying misconfigurations, and §1.3 reviews existing approaches to network specification. I then describe two new approaches, coverage-guided specification (§1.4) and relational specification (§1.5). I conclude with a summary of contributions in §1.6.

1.1 Network Configuration: Why Is It Difficult?

Each network has its own design and operational goals. For example, a cloud network must provide reliable connectivity to authorized user traffic while denying unauthorized ones. It must also enforce performance requirements, security regulations, and optimize resource utilization to the best possible extent.

The primary approach to achieve such goals is via router configurations. These are commands and parameters that controls how an individual router operates. In most modern networks, a router's main job is to coordinate with each other in a distributed manner[1] to determine how packets flow in a network (known as the routing process). This includes sending messages to neighbors to share information on available paths, as well as selecting the best path(s) based on each router's own preferences. Configurations can control many aspects of such routing processes. This includes which routing protocols are in use; which routers are peered with each other; how routing messages are transformed prior to sending (export policy) and upon reception (import policy); and the preference function for best path selection. Naturally, thus, how the network forwards packets is intimately dependent on router configurations.

Correctly configuring networks to achieve their goals has long been a challenging task for the networking community. While the goals are often global (*e.g.*, eliminating forwarding loops) and end-to-end (*e.g.*, ensuring traffic from node A to node D follows the path A-B-C-D), the configurations are localized to individual routers. Bridging this gap requires understanding the intricate interplay of various distributed routing protocols and maintaining a global perspective of the entire network. Compounding the challenge, configurations must be written in languages that operate at a very low level of protocol details and are specific to each vendor. Furthermore, real-world network goals are subject to frequent changes to adapt to dynamic business needs and operating environments. In light

[1]This is not the case for software-defined networking (SDN), a new paradigm of networking that governs individual routers using centralized, programmable controllers. This dissertation focuses on networks based on distributed routing, as the majority of today's networks are still based on this paradigm.

of such changes, engineers must update the configurations in a short time frame. Collectively, these factors contribute to frequent misconfigurations where human engineers' implementation diverges from their goals.

The fallibility of user-written configurations is a significant contributor to network reliability issues, as even seemingly minor errors can cascade into major disruptions with far-reaching consequences. For instance, in 2021, during routine maintenance of Facebook's backbone network, a single line of erroneous command unintentionally brought down all connections in the backbone network. This action resulted in the disconnection of all Facebook datacenters from the Internet. To make it worse, the loss of connection rendered Facebook's DNS servers unreachable, preventing engineers from accessing their work accounts to address the problem. Consequently, this global outage persisted for 6 hours before being mitigated, marking it as one of the most severe incidents in the history of the Internet. [58]

1.2 High-Quality Specifications: The Key to Catch Misconfigurations

Today, several approaches are employed to catch network misconfigurations. One common strategy is to continuously probe and monitor the performance and availability of the network [36, 69]. However, it only detects issues after they have already occurred, *i.e.*, it cannot prevent outages. Additionally, it fails to identify latent bugs that do not cause immediate problems but may later lead to widespread outages.

To proactively detect misconfigurations, common approaches include human review, testing, and verification. Similar to the standard practice of peer code review in software engineering, human review usually incorporates expert engineers to audit configuration contents and look for potential problems. While it does help sometimes, it does not fundamentally eliminate human errors. This statement is backed by the empirical observation that network outages continue to occur despite that human review has been employed as the standard practice in major cloud providers [59, 63].

A more systematic way to proactive debugging is automated testing. It involves run-

ning a set of user-defined test cases on a replica of the network (*e.g.*, a virtual replica simulating the network state after applying a planned configuration change) to check whether the network would violate the expected behavior specified by the test cases. Another approach is network verification [45, 46, 28, 13, 1]. Its goal is to establish a formal model of the given network state and check its compliance with formal properties expressed by users. For both testing and verification, users are responsible to write their expectations of network behavior (we use the term *specifications* to denote test cases or formal properties throughout this dissertation), and both approaches will only be effective when user-provided specifications are *complete* and *precise*. If the specifications are incomplete, they may fail to test the critical aspects of network behavior and leave bugs unnoticed. Specifications also need to be precise, which means it cannot be too loose nor too restrictive. For instance, a specification may be too loose if it focuses solely on basic connectivity between endpoints while neglecting the specific forwarding paths taken by the network traffic, as it would fail to detect bugs of suboptimal routing (which may cause high latency and unbalanced resource utilization). On the other hand, a specification may be too restrictive when imposing blanket requirements on all network devices without considering potential corner cases or network heterogeneity. This can lead to an overwhelming number of false alarms, making it challenging for engineers to distinguish genuine errors from benign deviations in test results. Overall, a high-quality specification for real-world networks should cover all traffic class carried by the network (for completeness); for each traffic class, it should assert whether it is forwarded along desired paths, and the desired paths in the specification must be specific and correct (for precision).

1.3 Existing Approaches to Network Specification

The creation and maintenance of complete and precise specifications is a difficult job given the vast scale of modern networks. So how well do existing specification methods do in helping battle the challenge? Surprisingly, no existing tool provides help in ensuring specification completeness, and it is prohibitively verbose to write precise specifications

API	Explanation
`bgpSessionCompatibility(nodes, remoteNodes)`	Test configuration compliance for BGP protocol.
`traceroute(startLocation, headers)`	Test forwarding paths for concrete packets.
`reachability(pathConstraints, headers)`	Verify reachability for symbolic packet headers.

Table 1.1: A subset of API functions provided by the Batfish tool.

in current specification languages. In general, there are three categories of specification methods.

The first one is to provide a parameterized function for each kind of network properties that can be checked by the tool. This is the case for the popular open-source tool, Batfish [12, 20], which exposes 73 Python functions for users to query different properties, ranging from configuration well-formedness, packet forwarding, to reachability, *etc.* Table 1.1 shows an example API function of each category. Network engineers are responsible of selecting the proper function and providing the correct parameters.

Another common approach is to use logic languages. For example, some tools require users to encode specifications as logic formulas [75, 13, 31]. Logic languages tend to be flexible and expressive.

The third notable approach is to adapt regular expression languages for network path properties [5, 15, 44]. A key insight is that network forwarding paths are akin to strings, where the former are sequences of network locations and the later are sequences of symbols. Therefore, regular expressions [37], the well-known tool for string pattern matching, can also be used to express network path patterns (after domain-specific augmentations).

1.4 Coverage-guided Specification

Existing specification languages are expressive and capable of specifying a diverse set of properties. However, one thing that they all lack is any sense of when the process of writ-

ing specifications is done (*i.e.*, complete). And that is a problem because in today's large and complex networks, it is surprisingly easy for engineers to overlook critical components or behaviors when creating specifications. This dissertation investigates a new approach, namely network coverage metrics, to alleviate that problem once and for all.

Network coverage help engineers judge how complete the network behavior has been captured by existing specifications, as well as identifying the gaps that remain unaddressed. Analogous to code coverage metrics [53, 4], which have been an integral part of any software testing platform [41, 19, 24], network coverage metrics aim to help users quickly and systematically locate under-tested aspects, so that users can engage in an efficient way of crafting specifications—adding specifications that are targeted to the under-tested aspects and progressively make the specifications more complete. As with software, high coverage is necessary but not sufficient for effective discovery of bugs. In addition to exercising all key behaviors, the specifications must also properly assert that those behaviors match their design goals (*i.e.*, specifications must also be precise). This latter task is not the focus of this approach but will be discussed in the next section.

For software, the development of coverage metrics and usable systems has been a multi-decade journey that still continues. In networking, however, we do not even have well-defined coverage metrics, let alone practical systems to compute them. We must first define network coverage metrics. We cannot simply reuse software metrics given differences in the two domains. Software may be viewed as a graph of basic code blocks, where each block is a linear sequence of statements. A simple yet effective coverage metric is the fraction of statements or basic blocks tested. On the other hand, network is a distributed system where devices exchange routing messages in configuration-defined ways to maintain distributed routing tables. Both the configurations (known as the network control plane) and routing tables (known as the network data plane) have semantics that differ from linear code statements.

Figure 1.1 illustrates the control plane and data plane of an example network. The two yellow boxes show a subset of configurations for routers R1 and R2. Here I use a syntax

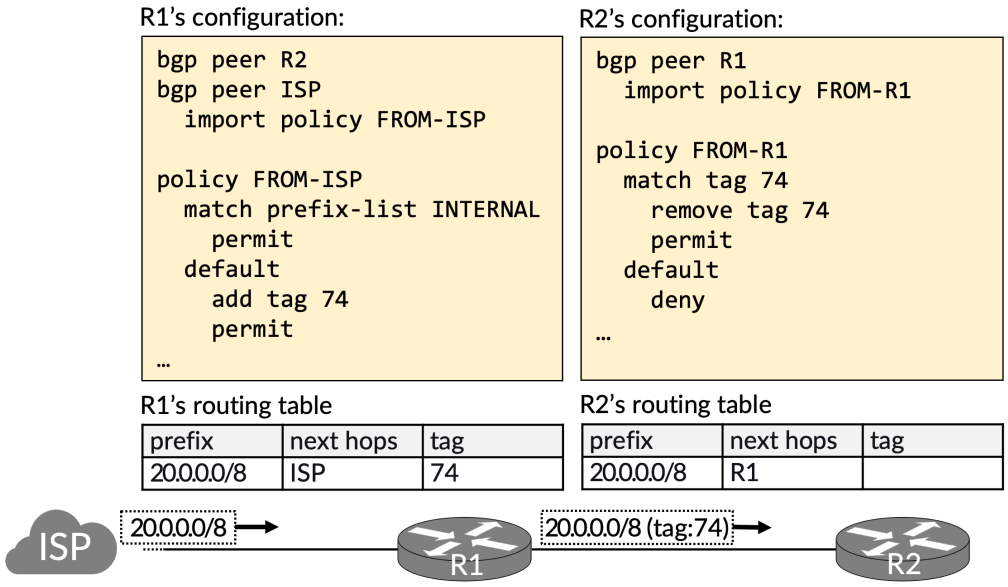


Figure 1.1: An illustration of network control plane (configurations) and data plane (routing tables). Dashed boxes denote routing messages.

simplified from Cisco [68]. The first section of both configurations defines the routing protocol that is used (BGP, Border Gateway Protocol), the peering relations, as well as the optional import policies used for each peer. The second section further defines each policy as a chain of MATCH-ACTION operations applied to routing messages. On the bottom of the figure, dashed boxes denote routing messages sent from ISP (Internet service provider) to R1 and from R1 to R2 respectively, and the routing rules derived from such messages are shown in each router's routing tables. Such tables are then used to instruct how routers handle network packets. When a packet arrives, the router will look up the packet's destination IP address to find the best routing rule in its table, and hence knows the desired next hop(s).

This dissertation proposes to evaluate the completeness of network specifications from the lens of both *data plane coverage* and *configuration coverage*. The former cares about how well different data plane elements (*e.g.*, forwarding rules, interfaces, and paths) have been

exercised by specifications; the latter reveals which configuration elements (a configuration line may contain multiple elements and an element may span multiple lines) have or have not been examined by specifications.

1.5 Relational Specification

Existing specification languages are useful in describing a single snapshot of networks. However, coming up with complete and precise specifications of large networks is essentially impossible, as one needs to enumerate the concrete packets and their desired forwarding paths for all traffic carried by the network. This makes it challenging to validate *network changes*, which are configuration changes made by engineers to adapt to new network traffic patterns, adjust security measures, or optimize resource utilization. To ensure that no network traffic is incidentally disrupted by a network change, single-snapshot specifications need to enumerate the precise forwarding behavior of all traffic that are not expected to change. Thus, precise single-snapshot specifications are proportional to *network size*, not the smaller *change size*, which makes them impractical to generate for many real-world networks.

This dissertation investigates a new approach to specifying network changes, namely *relational specification*. Rather than reasoning about the behavior of a single snapshot in isolation, relational network verification reasons about the similarities and differences (i.e., the *relationships*) between the behavior of *two* network snapshots.

Relational specifications make it easy to specify "no change" for the traffic and paths engineers do not want to modify (and may not even know about). Indeed, the size of a relational network specification is proportional to the complexity of the change rather than that of the network as a whole. If a desired network change is small (e.g., removing link A and shifting all of its traffic to link B), the relational specification will also be small. It is no wonder then that engineers already informally use such ideas to communicate their intent in change request tickets. In a sense, relational specifications capture formally the kind of thinking that engineers engage when making network changes, and in a way that

allows automatic checking.

1.6 Summary of Contributions

To summarize, this dissertation focuses on facilitating effective discovery of network misconfigurations. It achieves this goal by addressing the difficulty to write high-quality network specifications through two novel approaches: 1) coverage-guided specification and 2) relational specification. Our contributions are as follows:

- **Network data plane coverage.** We develop a general framework to define and compute network coverage for stateless network data planes. It computes coverage for a range of network components (*e.g.*, forwarding rules, interfaces, paths) and supports many types of tests and verification (*e.g.*, concrete versus symbolic; local versus end-to-end; tests that check network state versus those that analyze behavior). Our framework is based on the observation that any network dataplane component can be decomposed into forwarding rules and all types of tests ultimately exercise these rules using one or more packets.

- **A dependency model from network configurations to generated data plane elements**. To answer what configuration elements are covered by network tests that do not directly exercise configurations but exercise the data plane (*i.e.*, the output of configurations), we must establish the dependency between configuration elements and the data plane elements derived from them. We derive an information flow graph (IFG) model based on the semantics of network control plane that succinctly tracks dependency in the complex routing process of distributed control planes.

- **Network configuration coverage.** Based on the foundation provided by the IFG model, we define configuration coverage for both data plane and control plane tests. We also device an scalable algorithm to infer dependencies on demand, even when such dependencies are indirect or non-deterministic.

- **A formal specification language for network changes.** We develop a language called Rela to compactly describe the intent of network changes. Network engineers use regular expressions to represent paths and simple modifiers such as adding or removing parts of the path to specify change intent. Rela compiles this user-friendly language to a low-level, regular intermediate representation (RIR) based on the theory of regular languages and relations [37].

- **Relational network verification.** Given relational network specifications, we provide a decision procedure to verify that the pre- and post-change network snapshots adhere to the specification. Our verifier combines the compiled specification (RIR) with data from the pre- and post-change network snapshots, and checks that the pair of snapshots satisfies the RIR specification by reducing the problem to equivalence-checking for finite state automata.

Chapter 2

BACKGROUND

In this chapter I give an overview of network routing and forwarding in the context of contemporary industrial networks, as well as the common practice to check their correctness. §2.1 includes a description of commonly used forwarding and routing mechanisms such as destination-based forwarding, connected routes, Open Shortest Path First (OSPF), and Border Gateway Protocol (BGP). §2.2 overviews the current practice of network testing and verification, including the types of network properties that different types of tools can check, the specification languages, the simulation and emulation tools used to predict network behaviors, and the checking algorithms. Readers familiar with these concepts can skip ahead to the next chapter.

2.1 Network Fundamentals

Routing and forwarding are the two essential network functionalities that determine how data packets flow in a network. Routing is performed by what is called the *control plane* of the network. Its goal is to determine the best paths to deliver any data packets within the network. Based on the decisions made by the control plane, the *data plane* (also known as forwarding plane) then performs the actual forwarding when data packets arrive. Let us elaborate.

2.1.1 Data Plane and Forwarding

Forwarding refers to the process of transmitting data packets that enter a network device (such as a router) from an input interface to appropriate output interfaces. Different packets coming from different input interfaces are sent to different output interfaces based

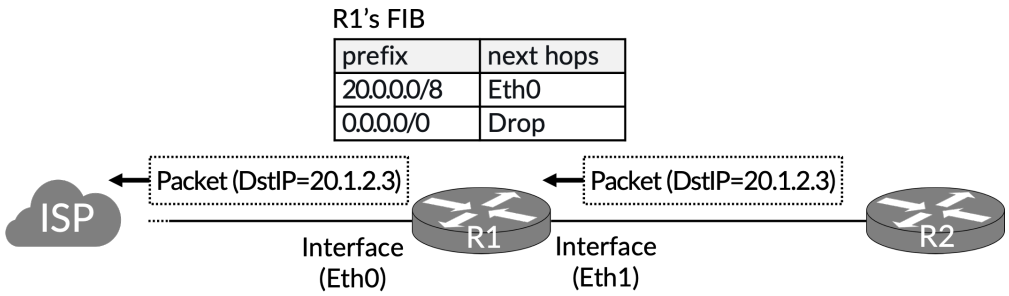


Figure 2.1: Example of destination-based forwarding and router's FIB data structure. When the packet (dashed box on the right side) arrives, the router examines the packet's destination IP address, finds the best FIB entry for this IP address, and then uses the next hop interfaces of this entry to forward the packet.

on the routing decisions made by the control plane. Such routing decisions, *i.e.*, a mapping from all possible combinations of packets and input interfaces to the desired set of output interfaces, usually need to be computed beforehand to facilitate fast forwarding. Therefore, network data planes are often implemented as lookup tables that stores this mapping.

The data planes of modern industrial networks are complex in two dimensions. Firstly, the format of lookup tables and corresponding lookup algorithms vary with different network protocols, and many networks run an increasing number of such heterogeneous forwarding mechanisms at the same time. Secondly, the scale of lookup tables themselves grows quickly as the networks become larger and carry more traffic. In the rest of this subsection, I will introduce the basics of commonly used forwarding mechanisms.

Destination-based forwarding. Conventionally, network devices that run the Internet Protocol (IP) make forwarding decisions solely based on the destination IP address carried in the packet header. Due to the large number of possible IP addresses (2^{32} for IPv4, 2^{128} for IPv6), it is impractical to store the mapping from each individual IP address to its intended output interfaces. Instead, the mapping is represented using a more com-

pact data structure called the forwarding information base (FIB), where each entry maps a destination IP prefix[1] to certain output interfaces or "drop". Figure 2.1 shows an example FIB of router R1. When a packet arrives from any input interface, the network device performs a lookup in its FIB to find the prefixes that contain the packet's destination IP address. A same IP address may be contained by the prefixes of multiple FIB entries due to the possible overlapping of IP prefixes. For example, $20.1.2.3$ in Figure 2.1 is contained by both $20.0.0.0/8$ and $0.0.0.0/0$. In such cases, FIB entries with the longest prefix among all possible matches will be preferred because they are the most specific to this IP address. Finally, the network device transmits (*i.e.*, forwards) the packet to the output interfaces dictated by the longest-prefix-matched FIB entries.

Label-based forwarding. Some modern networks adopt a new mechanism called MPLS (Multiprotocol Label Switching) in addition to traditional destination-based forwarding. It is often used to implement VPN (Virtual Private Networks) or differentiated service for different priorities. In MPLS, labels can be used to override destination-based forwarding. When an IP packet enters an MPLS-enabled network, an extended version of destination-based lookup is performed. In addition to finding a next hop, it may find a label that needs to be added to the packet. With the presence of labels, an IP packet becomes a *labeled packet* and its forwarding decision will be handed over to another lookup table called label forwarding information base (LFIB). Each entry in LFIB maps an incoming label to certain forwarding actions, which may include swapping the incoming label with an outgoing label, pushing a new label onto the packet, popping the label (a labeled packet is converted back to a normal IP packet when the last label is popped), and then forwarding the packet to certain output interfaces.

Policy-based forwarding. Policy-based routing (PBR) is yet another enhancement of IP networking. It enables static declaration of forwarding decisions as tiny programs (re-

[1]An IP prefix represents a range of IP addresses with the same starting sequences of bits, *e.g.*, $20.0.0.0/8$ denotes an IPv4 prefix with length of 8, which contains $2^{32-8} = 2^{24}$ unique IPv4 addresses starting with $0b00010100$. Specifically, prefix $0.0.0.0/0$ contains all 2^{32} IPv4 addresses.

ferred to as *policies*) in device configurations. Each policy can be applied to a specific input interface, so that all packets coming into this interface will be evaluated against this policy before other forwarding mechanisms step in. Such policies can examine various attributes of incoming packets (such as source IP address, packet size and protocol type) and apply various actions (such as directly indicating a next hop, dropping the packet, applying another policy, or handling over to another forwarding mechanism). With such flexibility, modern network data planes become more powerful and even more complex.

2.1.2 Control Plane and Routing

The data plane does not decide the forwarding next hops by itself. Instead, the network control plane makes such decisions in a process called routing. It involves the use of routing protocols to exchange path information between routers, select the best paths for each class of traffic, and build lookup tables to store such decisions. In this subsection, I will introduce the basics of distributed routing, the role of configurations in routing, as well as commonly used routing protocols.

Most modern networks run a distributed control plane, where each device (such as a router) runs one or more routing protocol instances. Each instance learns path information for different traffic classes via static configuration or dynamic message passing between neighboring instances. Such path information is stored in protocol-specific data structure to perform best path computation, which may involves running shortest path algorithms or applying certain tie-breakers to multiple available paths learnt from different sources. Once a protocol instance determines the best path for a traffic class, it will contribute this information to the routing information base (RIB) or label information base (LIB) of the router. The former stores destination-based paths, and the latter stores label-based paths. The information contained by RIB and LIB are then used to derive the mapping from input packets to forwarding actions, represented as the FIB and LFIB. Figure 2.2 illustrates this routing computation inside a router.

Configurations play a central role in regulating router behavior in distributed control

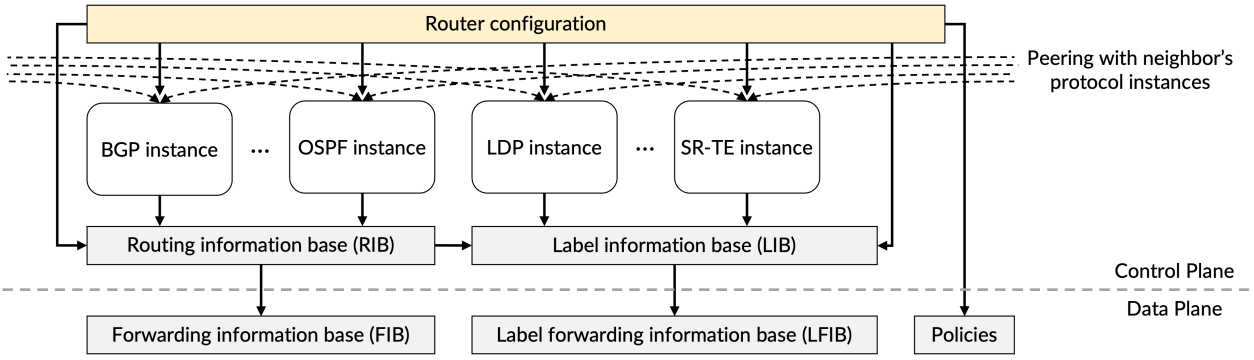


Figure 2.2: A zoomed-in view of routing computation inside a router. Rounded rectangles indicate routing protocol instances. The specific routing protocols that are run depend on the configuration. Solid arrows indicate the propagation of routing information inside a router. Dashed arrows indicate the peering relations and message passing between neighboring instances (the other sides are on neighbor routers and thus are not shown).

planes. This includes which routing protocols are run, what path information are owned by a routing protocol instance, with whom an instances is peered, which paths are to be shared, how to transform shared path information before being sent or after being received, and so on. Given the importance of configurations in achieving desired routing behavior, however, it is notoriously difficult to write correct configurations or to judge the correctness of given configurations. A handful of reasons contribute to this challenge. Firstly, the routing computation is distributed among possibly a large number of routers, and humans are not good at reasoning about distributed computations. Secondly, an increasing number of routing protocols come into use, each with different features and behaviors. Much expertise is required to understand various protocols and the interplay between them. Thirdly, the languages in which configurations need to be written have complex syntax that varies across router manufacturers. In the following of this section, I will introduce commonly used routing protocols and show example configurations for them in Cisco (one of the major router manufacturers) syntax. This introduction does not

mean to be comprehensive—there are numerous routing protocols being actively used and many have abundant features that cannot be covered in a chapter. We focus on a few protocols that are the most relevant to the approaches proposed in this dissertation. Hopefully, they are enough to show case the challenges described above.

Interfaces and connected hosts. When a host is attached to a network router through one of the router's *interfaces*, the router needs to know which IP addresses are used by the host. This can be done by statically associate an interface with a range of directly reachable IP addresses. Figure 2.3 shows an example. Assume router R2 is connected to a cluster of hosts via Ethernet1. The attributes of this interface can be specified using a section of configurations starting with "interface *interface-name*" (line 1). Then in line 2, following keyword "ip address", $1.0.2.1$ declares an IP address that is assigned to the interface itself, and $255.255.255.0$ declares a subnet mask with length 24. This line means that all IP addresses within the same $/24$ prefix as $1.0.2.1$ can be directly reachable via this interface. Based on this configuration, the router automatically generates an entries in its RIB. This RIB entry (or *route*, for simplicity) maps prefix $1.0.2.0/24$ to next hop interface Ethernet1, so that packets destined to attached hosts will be forwarded to this interface, as desired. The protocol field of this route usually appears as "connected" (or "direct" by some other manufacturers). Connected routes have the highest priority among routing protocols, so that the routes to local interfaces and directly connected hosts will always be preferred over routes learnt from other protocols.

Static routes. Network engineers can statically declare forwarding next hops to specific destinations using static routes. This approach is often used when forwarding paths for some traffic are fixed and known, so there is no need to compute routes dynamically. Figure 2.4 shows an example use case. The configuration keyword "ip route" is followed by IP address $20.0.0.0$ and subnet mask $255.0.0.0$, which collectively declares an IP prefix ($20.0.0.0/8$). Then, the following IP address $1.1.1.1$ indicates the next hop for this prefix. This configuration line will lead to an entry in R2's RIB with protocol field "static", as

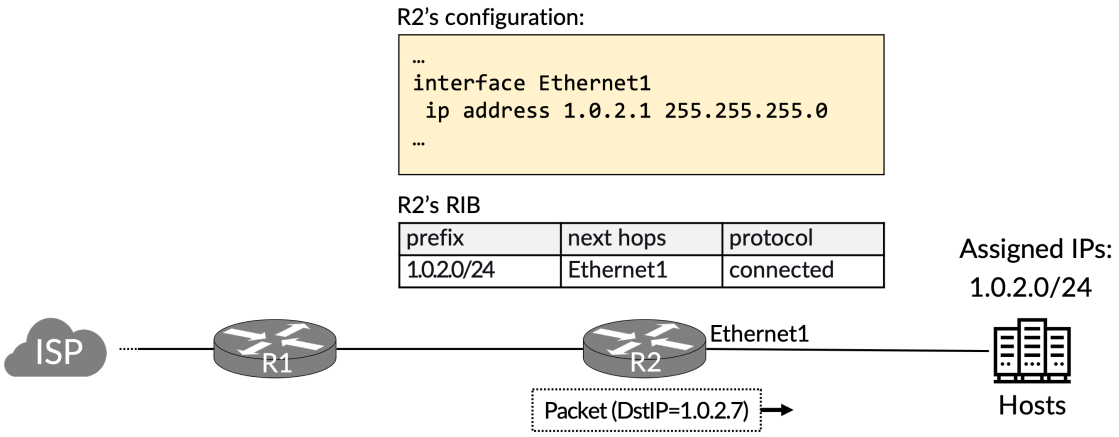


Figure 2.3: Example network using connected routes.

shown in the figure. Note that static routes can use IP addresses instead of interfaces to represent the next hops. In such cases, the actual output interfaces will need to be resolved by mapping the next hop IP addresses to their own next hops (*i.e.*, the next hops of next hops), using other RIB entries. If the derived next hops are still IP addresses, this process will continue recursively until an entry (such as a connected route) maps IP to interfaces. This resolution happens when translating the RIB entry into a FIB entry.

While connected interfaces and static routes can be useful for traffic with known forwarding paths, they need to be configured on a per-hop basis. Moreover, static paths make networks vulnerable to link failures. This leads to the adoption of dynamic routing protocols that can automatically propagate routing information from its source to the rest of the network or other networks and adapt dynamically to operating environments.

Open Shortest Path First (OSPF). One common routing goal is to make interfaces and directly attached hosts reachable from other routers in the same network. This can be achieved by having routers distribute routing information inside a network using interior gateway protocols (IGP). One example is OSPF, a *link-state* protocol that maintains a synchronized copy of link-state database (LSDB) on each router. Each router then inde-

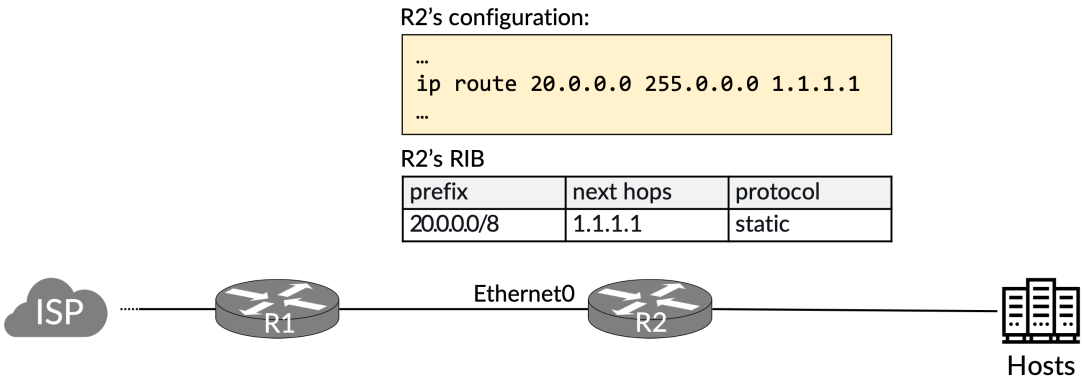


Figure 2.4: Example network using static routes.

pendently uses this information to compute the best paths to different destinations using Dijkstra's shortest path algorithm. The LSDB contains the state (up or down, and costs in both direction) for every link in the network, and it is synchronized by periodically flooding link-state advertisements (LSAs) throughout the network.

Figure 2.5 shows an example configuration of OSPF. Router R2 have directly attached hosts via Ethernet1. To inform the other router R1 about the available paths to these hosts, both routers enable the OSPF protocol using command "router ospf *instance-id*". The instance id is a number for identifying possibly multiple OSPF instances on a same router. The following commands "network *ip-address wildcard-mask*" then declare routes that needs to be shared within the network. Given a pair of IP address and wildcard mask, an IP prefix can be uniquely determined. The OSPF instance will then automatically find local interfaces whose connected routes belong to this prefix and try to establish OSPF peering through these interfaces. To allow routers learn the paths to each other's interfaces, the information of locally configured interfaces will be flooded to peers and the information of peer's interfaces will also be received. This gives each OSPF instance a global view of network topology as a connected graph, where each vertex denotes a router and each directional edge denotes the egress interface of a network link. Optionally, weights

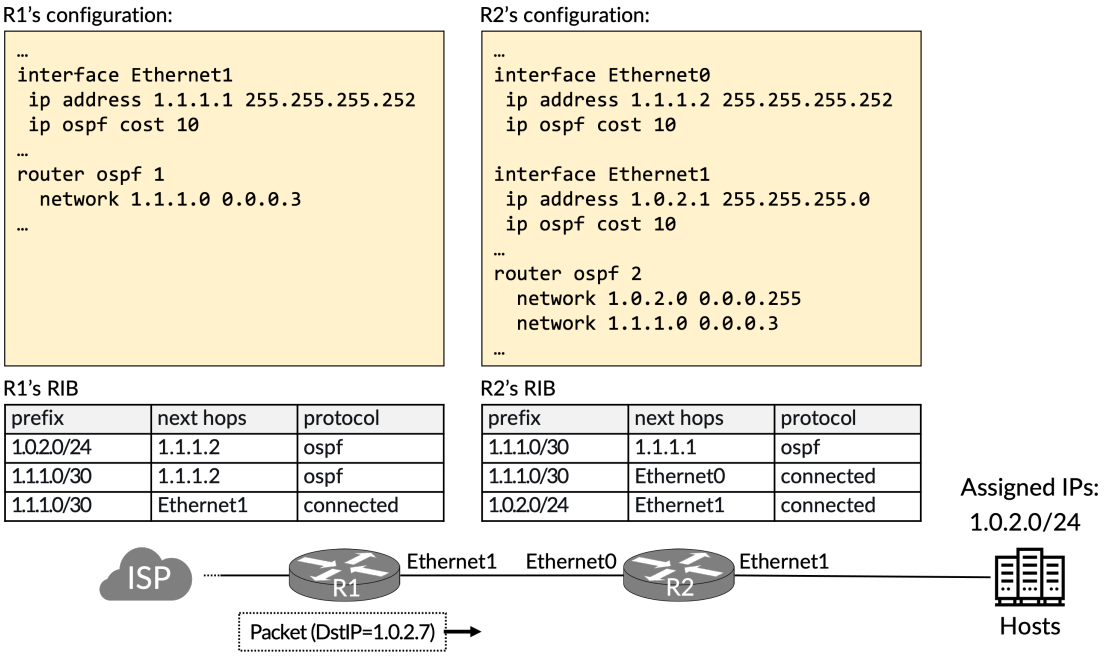


Figure 2.5: Example to establish interface connectivity inside network using OSPF.

can be assigned to each link using "ip ospf cost *cost*" command in the interface section of configurations. Using this information, each instance can find the weighted shortest paths to other routers' interfaces and store as OSPF routes in the RIB. In light of possible link failures, LSA updates are flooded to keep the LSDB up-to-date in all peers, and recomputation of best paths will be triggered. This allows routers to adapt to dynamics of the network and always select the best available paths.

Border Gateway Protocol (BGP). Interior gateway protocols such as OSPF can effectively establish connectivity inside a same administrative domain of network. However, the Internet consists of numerous networks independently operated by different organizations, and each network has its own path preferences based on security and economic concerns. To this regard, BGP, an exterior gateway protocol (EGP), became the de facto standard of routing protocol to exchange routing information between different autonomous systems (AS) in the Internet. Due to its better scalability compared to IGPs, BGP is also widely

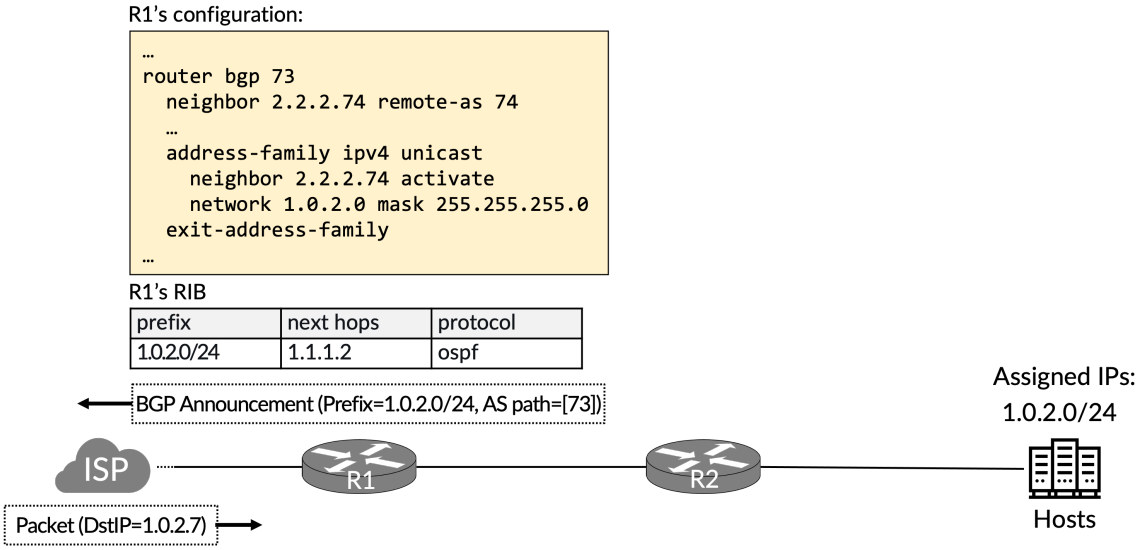


Figure 2.6: Example to share internal routes to the Internet using BGP.

used to establish internal connectivity in datacenter networks.

BGP is a *path-vector* protocol, where the routing messages (BGP announcements) encode the full paths (the AS number of each hop along the path) to different destinations. Each BGP instance announces its statically configured routes to its peers. Peers who learnt such paths can further announce them to their peers, and so on. Each instance then chooses the best paths for each destination by examining a list of *path attributes* such as local preference, path length, and multi-exit discriminator (MED). An attribute is checked only when all preceding attributes are tied among multiple paths. BGP allows flexible customization of message passing and path selection via *routing policies* set on the inbound and outbound of each BGP peer. These policies can apply various filtering and manipulations on path attributes. For example, a common routing policy used by Internet service providers is to set higher local preference for routes learnt from a *customer* compared to those learnt from a (lower-tier) *provider*. Thus ISPs gain economical benefits by preferring customer links (earning money) over provider links (paying money).

Figure 2.6 shows an example use of BGP configuration. Assume R1 and R2 have em-

ployed OSPF to establish interface connectivity inside the network, and thus R1 already learnt an OSPF route to R2's attached hosts, as shown in R1's RIB. If we further need the hosts to be reachable from the whole Internet, we can use BGP to announce this route to the Internet service provider that R1 is connected to. BGP supports multiple address families. In addition to the most widely used address family, IPv4 unicast, BGP peers can also exchange routing information for other address families in a same peering session. This includes IPv4 multicast, IPv6 unicast, VPNv4 and VPNv6, which are often used simultaneously by cloud networks. For different address families, the routing information to be shared and the handling of information are different. Correspondingly, BGP configurations are hierarchical, allowing fine-grained configuration for different address families. Let us look at R1's configuration, the top-level command "router bgp *as-number*" declares a BGP instance (or process, by BGP convention). Second-level commands with one indent are process-wide settings. For example, command "neighbor *ip-address* remote-as *as-number*" on line 2 provides the general information for a BGP peer and allows the peering session to be established. However, the peering does not exchange routing information until being explicitly activated in the third-level configurations—commands with two indents between "address-family *address-family*" and "exit-address-family". This example shows the address family settings for IPv4 unicast, which first enables IPv4 unicast exchanging with peer ISP using command "neighbor *ip-address* activate", and then declares an IPv4 unicast prefix to be announced to active peers using "network *ip-address* mask *subnet-mask*". Given this configuration, R1 will confirm the availability of routes to $1.0.2.0/24$ in its RIB before announcing it to ISP. This mechanism avoids sharing unavailable paths by accident. The BGP announcement will contain the AS path to the destination, which in this case, contains just one hop (AS number 73).

An additional example of using import and export policies to customize BGP announcement processing is shown in Figure 1.1 (using pseudo code instead of a specific manufacturer's syntax.)

2.2 Network Testing and Verification

The goals of network testing and verification are to check the correctness of networks, or being more specific to the context of this dissertation, the correctness of network routing and forwarding behaviors. But wait a second, how to define "correctness"?

Conceptually, the correctness can be defined as the compliance between the expected routing/forwarding behaviors and the actual routing/forwarding behaviors. To make it machine-checkable, there are three main problems to be cared about. 1) How to model and materialize the expected routing/forwarding behaviors, 2) how to obtain (or approximate) the actual routing/forwarding behaviors, and 3) how to check compliance between the two. This section aims to overview the current practice of network testing and verification regarding these three aforementioned aspects.

2.2.1 Specifications

This subsection concerns the first task to test or verify a network–modeling and materializing the expected behaviors of the network. We will first look at how to model the expected behaviors of networks using specification languages. In the later of subsection, we brief how specifications are written or materialized using such languages.

The expected behaviors of networks are ultimately determined by humans who design, build and operate them. Before automated testing and verification tools exist, network engineers communicate their personal understandings of expected network behavior with peer engineers via natural language documentations along with topology diagrams. Such informal forms of *specifications* cannot be rigorously checked by computers.

When building automated testing and verification tools, a more rigour form of specifications became necessary. The abstraction and language for such specifications very much depends on what properties can be checked by testing and verification tools. Broadly speaking, there are primarily two types of properties, namely *data plane properties* and *control plane properties*.

State-inspection properties
• Router R1's forwarding table must have the default route entry.
• Router R1's forwarding table must have an entry to prefix P with a next hop of neighbor.
• The access control list A1 on router R1 must have an entry that blocks packets to port 23.

Behavioral properties		
	Local	**End-to-end**
Concrete	• Router R1 must forward a given packet with destination D via neighbor N1. • Router R1 must drop a given packet with destination D and port 23.	• Ping between two endpoints must succeed. • Traceroute between two endpoints must traverse the firewall.
Symbolic	• Router R1 must forward *all* packets to prefix P1 via neighbor N1. • Router R1 must drop *all* packets to port 23.	• *All* packets in a defined set must succeed between two endpoints. • *All* packets between two endpoints must traverse a firewall.

Figure 2.7: Taxonomy of data plane properties with examples of each type.

Data plane properties. Data plane properties concern the state or behavior of a concrete network data plane. A taxonomy of data plane properties is shown in Figure 2.7. The first category (top table) is state-inspection properties that directly inspect elements of the data plane (such as FIB entries) and assert their existence or attributes. An example is to assert that a default route (route with prefix $0.0.0.0/0$) is present on all routers. In contrast, behavioral properties (bottom table) analyze the forwarding behavior of router or network.

An example is reachability property–expecting packets emitted from a source to in fact be able to reach a destination.

Behavioral properties can be further classified along two dimensions: *local* vs. *end-to-end* and *concrete* vs. *symbolic*. Local properties analyze individual devices, often by checking that a device forwards packets to a destination via certain interfaces. End-to-end properties reason about forwarding behavior across a series of devices. A concrete behavioral property checks the behavior of a single, concrete packet. A symbolic property might assert that *any* packet sent from a source will reach a destination—such properties concerns entire classes of packets. The terms "data plane testing" and "data plane verification" are sometimes used to discriminate concrete vs symbolic data plane properties. In this dissertation, we broadly refer to both categories of properties as "data plane tests", and use "concrete" and "symbolic" to distinguish between them.

Combinations of different types of tests are often used to specify the same network given their unique strengths and tradeoffs. State inspection tends to be faster; concrete tests tend to produce easier-to-understand results; symbolic tests tend to provide stronger guarantees; and local tests are more modular and efficient, while end-to-end tests provide better indication of whether high-level, network-wide invariants hold.

Control plane properties. Control plane properties concern the correctness of a snapshot of network configurations. This includes checking configuration content, analyzing the behavior of routing policies in the configurations, or analyzing the behavior of data planes produced by this configuration snapshot under different operating environments. The last category of control plane properties described above is usually checked by "control plane verification" tools to ensure correct network forwarding behaviors under arbitrary failures or external routing messages. They are similar to data plane behavioral properties except that they reason about the behaviors of multiple possible data planes produced by the same configuration. For example, "router R1 must be able to reach router R2 using packet p in data plane state N" is a data plane property, while "router R1 must be able to

reach router R2 using packet p under arbitrary up-to-k link failures in the network with configuration S" is a control plane property.

Specification languages. Given the diversity of different network properties, many tools ended up with providing parameterized functions as APIs to take specification input, where each function corresponds to a type of network property that can be checked by the tool. A collection of function calls with materialized parameters serve as the specification input for such tools. This is the case for the popular open-source tool, Batfish [12, 20], which exposes 73 Python functions for users to query different testing and verification properties, ranging from configuration well-formedness, packet forwarding, to reachability, *etc.* Table 1.1 shows an example API function of each category. Another common approach is logic languages. For example, some control plane verification tools require users to encode properties as logic formulas [75, 13, 31]. The third notable approach is regular expression languages [5, 15, 44]. Their insight is that network forwarding paths are akin to strings, where the former are sequences of network locations and the later are sequences of symbols. Therefore, regular expressions [37], the well-known tool for string pattern matching, can also be used to express network path properties (after domain-specific adaptations).

Writing specifications. The design of specification languages only determines what types of properties *can* be checked. The actual properties that *are* checked depend on what specifications are written using such languages. In the current practice of network testing and verification, the specifications are written by human engineers in a best-effort manner.

Given the vast scale and complexity of modern industrial networks, as well as the tight time frame within which engineers must complete the testing and verification procedure, three common strategies are used to write specifications that are as high-quality as possible. One is to selectively specify properties that engineers believe to be important. For example, when specifying network forwarding behavior using data plane state-inspection properties, engineers need to specify exactly which router, which destination prefix to be

inspected, as well as asserting a concrete set of next hops for the inspected entry. Manual enumeration of all FIB entries in this way would be infeasible for most industrial networks given the scale. Selective specification is often used to this regard, *e.g.*, when writing specifications to check forwarding behavior after adding a static route in a configuration change, one may resort to specifying only for the added route and leaving other entries in the FIB unspecified.

The second strategy is to write general properties that need to hold for almost all cases. For example, a common set of general properties are used to demonstrate the capability of control plane verification tools in almost all research paper on this topic, such as *no forwarding blackhole* and *no forwarding loop*. This is analogous to using "no crash" or "no infinite loop" properties in software verification. While they are useful to find some types of errors, general properties do not reflect the customized routing policies that are specific to each network, and thus can not adequately represent the expected behavior of networks.

The third strategy is to summarize the properties into patterns using topology symmetry, and then populate patterns into concrete specifications using topology metadata. A vibrant example is Microsoft's RCDC [42], which proposes that the expected forwarding behavior of contemporary Azure datacenters can be written as three patterns:

- All pairs of top-of-rack routers (the bottom-tier routers in the network topology who directly connect to hosts) should be able to reach each other.
- Traffic should always follow a shortest path.
- All redundant shortest paths should be available.

While this strategy can concisely specify major aspects of forwarding behaviors for some networks, not all networks have such strong symmetry, and many long-running industrial networks that were originally build symmetrically would accumulate heterogeneity and corner cases during multiple years of evolution.

2.2.2 *Network Emulation, Simulation and Formal Models*

Given written specifications, the next task in testing or verifying a network is to obtain the actual behaviors of a network. This is not simply probing and observing the behaviors of a live network, instead, we need to *predict* the behaviors of a network before it is deployed, such that misconfigurations and other types of errors can be caught before they hit the network infrastructure.

Three technologies can serve this purpose, each with its unique strengths and limitations. Network *emulation* [49] involves orchestrating router software images to create a virtual replica of a target network. Router images are run in servers, connected to each other using the same topology as the target network, and are applied the same configurations. Mock routing messages can also be injected from the borders of emulated networks. Emulated networks can precisely mimic the behavior of target network behavior under certain operating environments, and the testing toolchains for real networks can often be run as-is. The main drawbacks of this approach are the heavy consumption of computation resources and the unavailability of router images for some router manufacturers.

The second technology, network *simulation*, mimics network behaviors using an independent implementation of network control plane and data plane computations, and thus it does not depend on the availability of router software images and may consume less computation resources when engineered properly. The simulation of control planes and data planes are usually separate. Control plane simulators [28, 80, 55] take a topology, the configurations of each router, and operating environments (such as link up and down states and external routing messages) as inputs, mimic the routing computation as shown in Figure 2.2, and produce a data plane state. Data plane simulators [20] take a data plane state as input, which contains a topology as well as the forwarding tables and policies on each router. It then accepts queries regarding single-hop or end-to-end forwarding behaviors of concrete packets. The concerns of this approach is the engineering efforts taken to implement and maintain the rich features of various control plane and data plane

computations.

The third approach is to build a formal model of network behavior. This can be a data plane model or a control-and-data-plane model. A data plane model encodes a concrete data plane state into a symbolic representation of forwarding behavior [45, 79]. It enables data plane verification, *e.g.*, checking whether *any* destination IP in $1.0.2.0/24$ are forwarded along path R1-R2-hosts in Figure 2.5, given the current data plane state. Non-symbolic analysis would have to run forwarding queries (in simulation) or virtual traceroute (in emulation) once for each individual IP in the given range. A control-and-data-plane model encodes a concrete network configuration snapshot (a topology and the configurations of all routers) into a symbolic representation of both routing and forwarding behaviors. The operating environments are also encodes into the symbolic representation as well [13, 75]. It allows the reasoning of routing and forwarding behaviors under many possible environments all at once (often referred to as control plane verification), *e.g.*, checking whether the reachability between two routers always hold when any up-to-k links fail in the network, given a configuration snapshot.

2.2.3 Executing Tests and Verification

We briefly discuss how network testing and verification tools check the compliance between the expected network behaviors (the specifications) and the actual behaviors (modeled via emulation, simulation or formal methods).

Network emulation usually supports the checking of concrete data plane properties (such as running virtual traceroute or reading FIBs from routers) and concrete control plane properties (such as checking whether BGP peer sessions have been established). This is done by executing CLI commands in emulated router software and parsing the outputs. Similarly, network simulation supports checking concrete data plane and/or control plane properties by computing the control plane states, data plane states and forwarding behaviors. The data plane states produced by simulation or emulation can also be fed into data plane models to check symbolic data plane properties.

To check symbolic data plane properties (data plane verification), data plane models usually use an abstraction called forwarding equivalence classes (FEC). Each FEC represents a set of packets that have the same forwarding paths in a network. Recall that symbolic data plane properties concern the forwarding behavior of a range of packets. This can be done by finding all FECs that overlap with this given range and analysing their forwaring behaviors.

Lastly, symbolic control plane properties (control plane verification) are checked using formal methods such as satisfiability modulo theories (SMT) solvers and model checking. For example, when using SMT solvers, given the network model encoded as constraints N and control plane properties encoded as constaints P, we can use the SMT solver to check the satisfiability of constraints $N \wedge \neg P$. Property P always holds for network N if and only if $N \wedge \neg P$ is not satisfiable.

Chapter 3

NETWORK DATA PLANE COVERAGE

As discussed in introduction, outages can happen despite heavy use of testing when the users' specifications are incomplete and fail to test all important aspects of the network. In this chapter, we investigate whether we can help engineers write better specifications using data plane coverage.

3.1 Motivating Example

Consider the hypothetical data center network in Figure 3.1. It has a three-level hierarchy, with leaf routers at the bottom that connect to hosts (not shown), spine routers in the middle, and border routers at the top that connect to the wide-area network (WAN). The network runs the BGP routing protocol [47]. Each ToR has a prefix that it advertises inside the data center. The WAN announces the default route (0.0.0.0/0) to the border routers, which then propagate this route downward. The network is designed to withstand all possible single-node failures without disrupting application connectivity.

Assume that the intended connectivity invariants are that bidirectional connectivity must exist between each pair of leafs and between each leaf and the WAN. Three tests are in place to check these invariants. The first checks that each leaf can reach each other leaf using packets with destinations addresses in the right prefix. The second test checks that each leaf can reach the WAN using packets with destination addresses outside the data center prefixes. The third test checks that each border router can reach each leaf using packets with the right destination addresses. With these tests, the network engineers believe they have all their bases covered.

However, they may discover that they do not when router $B1$ fails and the whole data

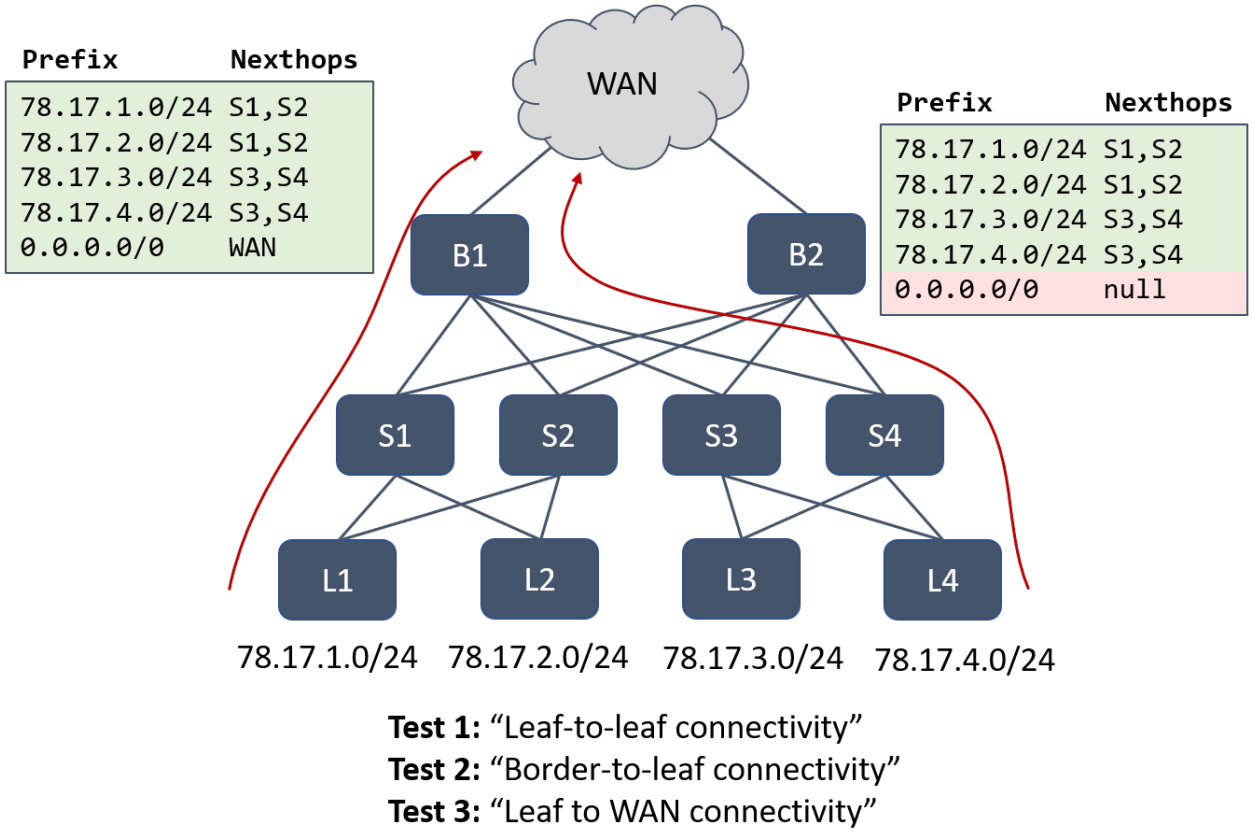


Figure 3.1: Test coverage for an example data center network. User tests check leaf-to-leaf, border-to-leaf, and leaf-to-WAN connectivity. Red arrows show the flow of packets from the leafs to the WAN. Forwarding table rules for B1 and B2 are shown, colored green if covered by a test and red otherwise.

center gets disconnected from the WAN, despite $B2$ being alive. It turns out $B2$ had a static default route that was null routed, which caused it to not propagate the default route to spines. So, when $B1$ fails, the spines have no route to the WAN.

How could the engineers in this scenario have uncovered the issue with their tests and prevented the outage? Once you know the exact root cause, many solutions suggest themselves. But instead of focusing on individual bug types, we need a general approach to uncover gaps in testing.

Coverage metrics can be the basis for that general approach. Suppose the engineers could compute *rule coverage* of their tests. Informally, this metric is the fraction of rules in the network through which least one test packet will pass; we formally define this metric later. Now, because no test packet uses the default route on $B2$, the coverage metric for $B2$ would have flagged the problem. It would have been lower than expected and also lower than $B1$, $B2$'s symmetrically-configured counterpart. The coverage metric could also have helped engineers improve the test suite. Once the underlying problem is discovered, the engineers could modify the test to not only check that the WAN is reachable from leaves but also that *all* spines and borders serve as conduits for this traffic.

3.2 Metric Computation Requirements

Network coverage metrics are intended to provide a sound basis for judging how well different network components such as devices, interfaces, and routes are exercised by the test suite. Exercising the component does not mean that all related bugs are caught. A test suite has two activities: i) exercising some components; and ii) asserting that resulting behaviors match expectations. To find bugs, the test suite must do both well. We focus on quantifying the quality of the first activity. This focus is similar to the software domain, where coverage metrics tend to quantify the fraction of statements or files exercised by the test suite and leave the task of judging the quality or assertions to other means.

3.2.1 *Support diverse metrics and tests*

We seek an approach to computing network coverage metrics that can support a diverse set of settings. Diversity refers not only to the types of networks (data center, backbone, etc.) and devices, but also to the types of metrics and tests. Let us elaborate.

Support for diverse metrics. For software, many different metrics have been devised to quantify coverage, such as the fraction of statements covered, the fraction of subroutines covered, the fraction of branches for which both paths are evaluated, and the fraction of control flow edges covered. Different metrics help software engineers focus on different aspects of coverage—for a given test suite, one metric may be high and another low, revealing a systematic testing deficiency. Different metrics may also represent different trade-offs in computation cost and bug-finding ability.

Networks too may be analyzed from many different perspectives. An engineer might ask: Do our tests cover every device? Every interface? Every path? Every flow? What do they say about a particular pod or about leaf routers? Each such question sheds light on a different type of testing gap. In the example above, we saw how rule coverage could identify the testing gap. This gap, however, would not have been revealed by device coverage, or the fraction of devices traversed by a test packet. Device coverage would have been 100% because every device was being traversed by at least one test. This includes $B2$, which was being covered by the border-to-leaf connectivity check (Test 2 in Figure 3.1).

Thus, our goal is not to devise a perfect coverage metric but support computation of a broad range of metrics. Doing so will enable network engineers to ask many different questions of interest and to drill down and investigate testing gaps.

Support for diverse tests.

Network engineers use several types of data plane tests. As shown in Figure 2.7, the tests can be broadly classified into *state inspection tests* or *behavioral tests*. State inspection tests directly inspect elements of the forwarding state and check that it matches expectations. An example is a test that checks whether the default route is present on a router. In

contrast, behavioral tests analyzes the device or network behavior. An example is a test that executes a traceroute and then verifies packets emitted from a source can in fact reach a destination.

Behavioral tests can be further classified along two dimensions: *local* vs. *end-to-end* and *concrete* vs. *symbolic*. Local tests analyze individual devices, often by checking if a device forwards packets to a destination via certain interfaces. End-to-end tests reason about network behavior across a series of devices.

A concrete behavioral test such as a traceroute checks the behavior of a single, concrete packet. A symbolic test might check if *any* packet sent from a source will reach a destination—such tests reason about entire classes of packets. The terms ”testing” and ”verification” are sometimes used for these categories. In this paper, we use ”test” to refer to all types of tests, and use ”concrete” and ”symbolic” to distinguish between the categories.

Multiple test types are often used for the same network because different types of tests have different strengths. State inspection tends to be faster; concrete tests tend to produce easier-to-understand results; symbolic tests tend to provide stronger guarantees; and local tests are more modular and efficient, while end-to-end tests provide better indication of whether high-level, network-wide invariants hold.

Coverage computation must thus support diverse types of tests. Test diversity poses a challenge, however. A basic function of a coverage metric is to enable users to judge if a new test will improve coverage compared to existing tests. This judgement is easy within the context of the same test type. For instance, compared to existing traceroute tests, it is easy to tell if a new traceroute test will exercise new network components, based on whether it uses a different source or packet headers. But it is hard to tell if a new traceroute test adds value compared to, say, existing symbolic tests.

3.2.2 *Properties of metrics*

To help users quantify and improve the quality of test suites, while supporting diverse networks, metrics, and tests, we devise metrics with the following properties.

Compositional. To compute multiple metrics for multiple test types in a tractable fashion, the metrics must be compositional in two ways. First, to seamlessly support diverse test types, equivalent sets of tests should yield equivalent coverage measures. Hence, we demand i) The coverage of a symbolic test must equal the combined coverage of a collection of concrete tests that collectively cover all those packets; ii) The coverage of a state-inspection test must equal the coverage of a symbolic test that considers all packets that can be impacted by that state. Second, we should be able to compute end-to-end coverage metrics (*e.g.*, for network paths) by composing the coverage metrics for a set of local tests, and compute local coverage metrics (*e.g.*, for individual devices) by decomposing what is tested by end-to-end tests.

We support such compositionality by mapping tests to a set of pairs of packet and forwarding state entry, a representation that is independent of the test type. We then compute all coverage metrics by taking a union of such sets. This uniform representation enables consistent treatment of different types of tests and prevents any double counting when multiple tests cover overlapping forwarding state entries.

Semantics-based. Network coverage metrics should be based on the semantics of network state and independent of how devices process the state. Different devices may have different implementations for processing state. Take longest-prefix matching (LPM) as an example. One device may linearly scan the forwarding table (FIB) sorted by prefix length to find the matching entry, and another may use prefix-tries. Analogous to the software coverage metric of fraction of lines exercised by a test, we may consider the fraction of FIB entries inspected as our coverage metric. Per this metric, for a test packet that matches the default route (0.0.0.0/0) entry, the scanning-based device would have touched all the FIB entries and the trie-based device would have touched a handful of entries. Such de-

vice implementation-based metrics are undesirable as they are unlikely to be aligned with network engineers' expectations. In fact, the engineers may be completely unaware of internal device implementations. We thus develop metrics that are based on semantics of network state. For the case of a test packet that matches the default route entry, we should deem that only that entry is exercised.

Monotonic and bounded. Finally, the metrics should have certain basic numerical properties. In particular, they should be monotonic and bounded. A metric is *monotonic* if adding a test to an existing test suite never causes the metric to decrease. Formally, a metric Cov is monontonic when for all test suites T and individual tests t:

$$Cov[T] \leq Cov[T \cup \{t\}]$$

In other words, adding a new test never diminishes the value of a test suite (though it may not always strictly increase the value).

A metric is *bounded* if it varies between a well-defined minimum (0) and maximum (1). The minimum should correspond to the case where no tests are performed, and the maximum to the case where no further tests can possibly increase the value of the test suite. Boundedness helps the user gauge how far their current test suite is from full coverage, and it also helps compare coverage across different networks and across time for the same network.

Monotonicity and boundedness together imply that users can increase coverage by adding appropriate tests to a test suite whose coverage is currently less than ideal. And if test suite coverage does increase, it covers more of the network state, thus increasing the probability of uncovering more bugs in the system.

3.3 Defining Network Coverage

Our network coverage metrics are based on general models of the network forwarding state and tests. We first describe these models and then our coverage computation framework. Figure 3.2 summarizes our notation.

Notation	Description
r/R	A match-action rule/rule set.
$S[v]$	The set of rules associated with device v.
p/P	A located packet/packet set.
t/T	Network test/test suite.
$M[r]$	The match set of the rule r.
$F[r][p]$	The resulting packets after applying rule r on packet p, where $F[r]$ is the action of the rule r.
$T[r]$	The set of packets used to exercise r by test suite T.
$P \triangleright r_1, \ldots, r_j$	Guarded string, describing a flow of packet set P along the path $r_1, \ldots, r_j$, see §3.3.3.
(P_T, R_T)	Coverage trace of test suite T, a tuple of a packet set P_T and a rule set R_T, see §3.4.2.

Figure 3.2: Network coverage concepts and notations.

3.3.1 Network Model

A network N is a 4-tuple (V, I, E, S). V is the set of network *devices*, and I is the set of *interfaces* on those devices. A network *location* ℓ is a pair, written $v.i$, of a device v and interface i. The set E contains links that connect locations. Finally, S, represents network forwarding state. For simplicity, each device contains a set of *rules* (R). We write $S[v]$ for the set of rules associated with a device v. A more sophisticated device model will have multiple tables of rules (*e.g.*, forwarding rules and access-control rules). Such extensions are straightforward but not necessary for explaining the coverage concepts that are our focus.

Rules operate over *located packets* (p), which include their location (ℓ) as well as the contents of their header fields. We use P to denote a set of located packets. A rule r will match some set of packets, called its *match set*, and apply an *action* to modify the matched packets. We write $M[r]$ for the match set of the rule r and $F[r]$ for the action of r. The match sets of rules do not overlap, making the rule that applies to any packet unambiguous. In practice, rules are ordered within tables and their match fields may

match overlapping packet sets. The first rule in a table to match a packet is applied to the packet. In our model, such rules have been preprocessed to eliminate overlapping match sets; our implementation computes these match sets.

Possible actions of a rule include forwarding the packet via one or more interfaces, dropping the packet, and transforming some subset of header fields. In general, $F[r][p]$ is a set of located packets P. If P is empty, the input packet p has been dropped. If P contains a single located packet p', the input packet has been forwarded and possibly transformed. If P contains multiple packets on different interfaces, the device has multicast the original packet. To apply an action to a set of packets P, we apply the action to all packets $p \in P$ and take the union of the results.

Model limitations. Our model has static view of the network forwarding state and assumes that all tests were run on this state. This view is consistent with how all data plane verification tools operate—they take a snapshot of network state and run all queries on it. But it may lose precision for live network tests (*e.g.*, ping or traceroute) if the state changes during test suite runs.

3.3.2 Modeling Network Tests

A test, be it a concrete traceroute or a symbolic analysis, checks whether the network handles some packets correctly by 1) consulting the network forwarding state, 2) computing packet transformation and forwarding, and 3) comparing the result with user-specified expectations. Coverage is determined by analyzing the forwarding state consulted in the first step and the set of packets considered in the second step. An Atomic Testable Unit, or ATU, (r, p) is our primitive measure of coverage—it indicates a test has exercised rule r on located packet p.

We model a test t as a (total) function from rules to sets of packets. When $t[r] = P$, we say that the test t has exercised rule r using packets P. Alternatively, we say that test t *covers* packets P for rule r. When $P = \{p_1, \ldots, p_k\}$, the corresponding set of ATUs is

$\{(r, p_1), \ldots, (r, p_k)\}$. Note that the set of packets P cannot exceed the match-set of r.

The *covered set of a test* is the union of all packets covered for any rule r. That is, the covered set for t is $t[r_1] \cup \cdots \cup t[r_k]$ when $r_1, \ldots, r_k$ is the set of all rules in the network. Likewise, the ATUs for t are $\{(r, p) \mid v \in V, r \in S[v], p \in t[r]\}$.

To illustrate these concepts, when $t[r]$ is the empty set, r has not been exercised by the test at all. When $t[r]$ equals the match set of r, r has been completely tested. Often, $t[r]$ will be somewhere in between, indicating that a rule has been partially tested. For example, when representing a traceroute test, $t[r]$ will be $\{p\}$ for some packet p for each rule r along the traceroute path and empty set for each rule r not on the path. When representing a symbolic test, $t[r]$ may consist of many packets rather than just one.

A test suite T is simply a set of tests: $\{t_1, \ldots, t_k\}$. In a slight abuse of notation, we often treat test suites as functions from rules to packets. Applying a test suite to a rule yields the union of the packets tested by all tests in the suite:

$$T[r] = t_1[r] \cup \cdots \cup t_k[r]$$

The ATUs for a test suite are defined in the obvious way as the union of the ATUs of the underlying tests in the suite.

Model discussion. An ATU represents the finest granularity of the impact of a test. An alternative is to use rules as atomic units. However, that would have rendered the coverage framework unable to distinguish between a concrete test, like a traceroute, that exercises a portion of the rule with a single packet and a symbolic test that exercises more of the rule over many packets.

A limitation of our model is that it cannot account for coverage of stateful networks accurately. If a rule r uses a switch register or another stateful component, exercising it once on a particular packet may not suffice to test it completely. Since ATUs (r, p) do not track the state space covered by applications of r to packets p, our coverage metrics will be blind to the amount of the state space covered. We followed common software-testing frameworks in making this choice. They too typically measure coverage in terms

of lines of code rather than state spaces covered per line of code. In both networks and software, tracking the state space covered may be prohibitively expensive. Thankfully, many network data planes are stateless, obviating the need to track state in such contexts.

3.3.3 Coverage Metrics

Given the models of the network state and tests, we can now define coverage metrics. Given our goal of supporting a diverse range of metrics, we developed a common framework for computing coverage for a variety of network components such as rules, devices, and paths. Below, we first describe this framework and then how different components map to it.

A framework for computing coverage

Our framework computes the coverage of an individual component (*e.g.*, a device or rule); the coverage of multiple components of interest can then be aggregated (§3.3.3). The specification to compute the coverage of a network component has three parts:

- a *dependency specification* G,
- a *coverage measure* μ, and
- a *combinator* κ.

The dependency specification G describes the dependencies of a component—what needs to be tested to test that component. Specifically, $G = \{g_1, \ldots, g_k\}$ is a set of *guarded strings*, where a guarded string $g = P \triangleright r_1, \ldots, r_j$ is a located packet set P followed by a list of rules $r_1, \ldots, r_j$.[1] The packet set P is the guard. The network engineer is interested in

[1]Guarded strings are a natural unit of interest in network data planes. They were also used to provide semantics to NetKAT expressions [5].

the flow of those packets along the path $r_1, \ldots, r_j$ where r_1 through r_j are rules that form a valid path (i.e., r_i forwards to r_{i+1}).

If the engineer is interested in understanding coverage of a single device with three rules then the dependency specification might be $\{P_1 \rhd r_1, P_2 \rhd r_2, P_3 \rhd r_3\}$. In this case, each "path" is only one rule as long as we are not interested in multi-rule paths. The packet spaces P_1, P_2 and P_3 might be the match sets of the rules.

The coverage measure μ evaluates the extent to which a test suite T covers a guarded string g in G. It must be a function from T and g to a number between 0 and 1, with higher values implying higher coverage. For instance, given $g = P \rhd r$, a measure might return 1 if there exists a test that exercises the rule r on a packet $p \in P$ and 0 otherwise. Alternatively, a measure might return the ratio of packets $p \in P$ that a test exercises on r. When considering a path $g = P \rhd r_1, \ldots, r_j$, a measure could determine the minimal fraction of any rule's match set in the path covered by a test.

Finally, the combinator κ produces an overall coverage metric for a component by mapping sets of measures to measures. Such combinators might compute the average of the given measures or use the min or the max value to report coverage for the component; as discussed below, different choices are appropriate for different sorts of component.

Formally, given combinator κ, measure μ, and specification G for a component, the coverage of test suite T for that component is:

$$CompCov[T](\kappa, \mu, G) = \kappa\,(\mathsf{map}\,(\mu[T])\,G) \tag{3.1}$$

The function $\mathsf{map}(f)(S)$ applies f to every element of the set S.

Since network operators are interested not only in coverage of a single device or flow, our framework also defines coverage over collections of components (e.g., all devices in the network). This coverage of is programmable as well. Given an *aggregator* α and a specification of the collection $C = \{(\kappa_1, \mu_1, G_1), \ldots, (\kappa_k, \mu_k, G_k)\}$, we can define the overall coverage of a test suite as follows.

$$Cov[T](\alpha, C) = \alpha\,(\mathsf{map}\,(CompCov[T])\,C) \tag{3.2}$$

§3.3.3 illustrates the use of Equation (3.1) by providing concrete specification for common network components, and §3.3.3 describes some useful aggregation functions for Equation (3.2) to summarize coverage for multiple components.

Coverage for common network components

We show how to analyze coverage for the most common network components: rules, devices, interfaces, paths, and flows. Other components, such as the CoFlows [22] of a distributed application (*i.e.* the set of flows generated by the application) or all traffic traversing a firewall, can be analyzed similarly.

Rule coverage. Rule coverage quantifies the extent to which a network rule is covered by a test suite. Given a rule r, the dependency specification is $G = \{M[r] \triangleright r\}$. The coverage measure adopted is $\mu = \frac{|T[r]|}{|M[r]|}$, which quantifies the fraction of the rule's match set covered by the test suite. This ratio will always be less than one because $T[r] \subseteq M[r]$. The combinator κ for this metric simply picks the only element in this singleton set.

Device coverage. Device coverage quantifies how well the forwarding state of the device is covered. Given a device with k rules, its dependency specification is $G = \{M[r_1] \triangleright r_1, \cdots M[r_k] \triangleright r_k\}$. Device coverage uses the same measure μ as rule coverage. Its combinator κ is the weighted average, where the weight is proportional to a rule's match set, that is, the weight for r_i is $\frac{|M[r_i]|}{\Sigma_{1 \leq j \leq k}|M[r_j]|}$. This way, device coverage reports the fraction of total packets against which the device as a whole has been tested.

Interface coverage. Interface coverage quantifies how well the state responsible for packets leaving or entering an interface is tested. For instance, engineers may evaluate outgoing interface coverage of border interfaces when they are interested in packets leaving the data

center. The coverage specification for an interface is similar to that of a device except that the set of rules considered is limited to those relevant to the interface. For an outgoing interface, this set has all the rules that forward packets to the interface. For an incoming interface, it has all the rules that have the interface in their match sets, as we limit the corresponding packet guards to only those on the interface.

Path coverage. A path is a valid sequence of rules and path coverage is intended to quantify how well the forwarding state along a path is tested. The dependency specification of the path is $G = \{P \triangleright r_1, ..., r_k\}$, where $r_1, ..., r_k$ is the sequence of rules that define the path, and P is the full set of located packets that can traverse the path. This set is not known apriori but can be compute by processing the forwarding state. It includes packets whose headers may get transformed along the way. It does not include packets that are dropped at an intermediate rule $r_{j<k}$ in the sequence; those packets will be part of the $r_1, ..., r_j$ path.

The measure μ of a path quantifies the fraction of P that has been tested through the entire sequence of rules. If different rules of the path were tested using disjoint sets of packets, the coverage will be zero, as no one packet has made it all the way across the path. This calculation proceeds rule by rule and computes at each hop the set of packets that remain under consideration. Formally:

$$P_i = \begin{cases} M[r_1] & \text{if } i = 0 \\ F[r_i][P_{i-1} \cap T[r_i]] & \text{otherwise} \end{cases} \tag{3.3}$$

P_i is the set of packets after rule r_i has been processed. In the beginning, it is $M[r_1]$ (match set of r_1). After processing r_i, it changes to the set obtained by applying the action of r_i to the intersection of P_{i-1} and r_i's tested set. The final coverage is $\frac{|P_k|}{|P|}$, the fraction of packets left at the end.[2]

[2]We have presented a simplified view that assumes that any packet transformations are one-to-one, so rule application preserves the number of packets in the set. To support one-to-many and many-to-one transformations, our system does a slightly more complex computation. It computes another sequence of packet sets P'_i that is not constrained by $T[r_i]$, by replacing it with $M[r_i]$ in the equation. It then computes $\frac{P_i}{P'_i}$ at each hop and takes the minimum ratio across all hops.

Paths have only one guarded string, and hence κ selects the only element of this singleton set as its result.

Flow coverage. A flow can be defined using its source location and header space. When this flow is injected into the network, it traverses one or more paths (due to possible multi-path routing or different headers in the header space being forwarded differently). The dependency specification of the flow has a guarded string corresponding to each of its paths, and each such string has the flow's packets as the guard. The coverage measure is the same as that for a path. The combinator is weighted average, where the weight is proportional to fraction of flow's packets that will use a path, computed by processing the network state. We are thus computing the fraction of flow's dependencies that have been tested end-to-end. If the coverage is 75%, it means that state corresponding to 75% of the flow's packet stream has been tested end-to-end.

Coverage for component collections

We illustrated above how to compute coverage for individual components, but users will commonly be interested in aggregate coverage across multiple components of the same type. We currently support three forms of aggregator α, each of which provides a different perspective on how well components in the collection are covered.

• *Simple average:* Compute the (unweighted) mean coverage across components. Average device coverage will thus be the mean of the coverage of all devices under consideration. This aggregation provides a simple, overall view of how well devices are covered.

• *Weighted average:* Compute the weighted average coverage across all components, where the weight is the size of packet space handled by the component. This aggregation gives a higher weight to components that handle more packets. Rules that match more packets (e.g., the default route) will thus have a higher weight, encouraging engineers to cover them with a higher priority.

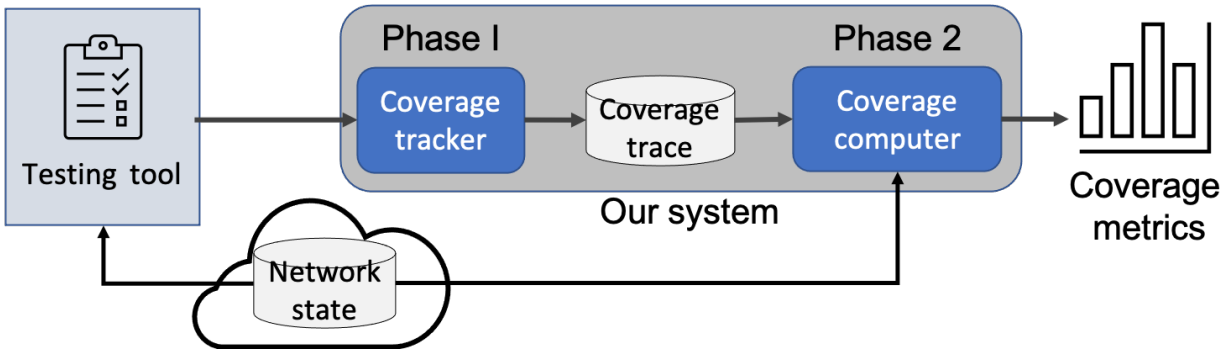


Figure 3.3: Overview of Yardstick.

• *Fractional coverage:* Often we simply want to know what fraction of components have been tested *at all*. The fractional coverage aggregator maps the coverage of individual components in the collection to 0 or 1, based on whether the coverage is 0 or non-zero, and then computes the mean. If fractional device coverage is 80%, then 20% of the devices are completely untested and users would likely want to investigate why.

3.4 Yardstick Design

Our system, Yardstick, is based on the formulation above. Its design is guided by the need to minimize impact on testing performance. This goal is important because some testing tools are in the critical path of network updates [46, 44] and testing delays will delay updates. In other settings, testing performance dictates how quickly a network state bug will be caught after it appears.

With this consideration, we split the operation of Yardstick across two phases, shown in Figure 3.3. The first phase tracks coverage reported by the testing tool, using two simple API calls:

- `markPacket(P)`
- `markRule(r)`

The `markPacket` call is used for behavioral tests, to report the located packets P used in tests. The `markRule` call is used for state inspection tests, to report which rules in the network are inspected.

The coverage tracker stores the information provided by testing tools in a compact representation, called the *coverage trace* (§3.4.2). The format of the coverage trace enables memory and time efficiency for both tracking and computing phases.

After testing finishes, in the second phase, Yardstick uses the coverage trace and network state to compute the requested coverage metrics. Since all the data is available, the network engineer can at any time ask the system to compute new metrics and zoom in from aggregate to individual component metrics.

Yardstick can be used with any testing tool that can be instrumented to call its APIs. Originally, we attempted a design that did not use explicit APIs such as `markPacket`, but snooped state read operations and registered coverage for elements of the state read during testing. This design assumed that all state elements read by the tool are exercised. Unfortunately, however, symbolic testing often requires that state be read in bulk up front to build internal data structures. It may not necessarily exercise all the data read.

Our chosen APIs enable easy integration with a range of testing tools. The information they need is readily available. Yardstick takes on the work to translate this information into what is needed for coverage computation. We discuss this next.

3.4.1 Using Coverage Tracking APIs

Different types of tests described in §3.2.1 can use our APIs to report coverage information with minimal overhead.

State inspection tests. These tests inspect forwarding state rules, *e.g.*, check whether a default route exists on a device. The coverage of inspecting a rule is its match set, so the information we need for metric computation is $t[r] = M[r]$.

However, $M[r]$ is not readily available to the testing tool (because the match field of

the rule is not necessarily its match set, given earlier overlapping rules in the table) and it can take time to compute for large tables. Yardstick thus requires tools to just report the tested rule via `markRule(r)`, and it then computes $M[r]$ in the second phase to minimize the burden on the testing tool.

Local behavioral tests. These tests inject a located packet set P into the device and then validate that the device processes (drops or forwards) the packets as expected. Different testing tools compute the result of the device processing differently; it could be concrete simulation, symbolic simulation, or SMT constraints. Regardless of the method, the information we need from the perspective of coverage, is which rules were exercised using which subsets of P (given P may exercised multiple rules).

Unfortunately, this information is not computed by all tools as part of their testing, and computing it can be expensive. Yardstick thus requires the tools to just report P using `markPacket(P)` and derive the rule-level coverage information from it later.

End-to-end behavioral tests. Coverage for these tests is reported using `markPacket(P)` as well but a separate call is made for each hop in the network with the packet set at that hop. We did consider an alternative in which testing tool would report the packet set at only the origin instead of each hop and we would then infer the hop-level information. But we deemed that alternative not worthwhile because the tools already have hop-level information available.

3.4.2 Computing Coverage Metrics

During test execution (Phase 1), Yardstick stores the union of all information reported by the testing tool in the coverage trace. The trace can be represented using the tuple (P_T, R_T), where P_T is the set of located packets across all `markPacket` API calls, and R_T is a set of rules across all `markRule` API calls. Yardstick does not keep the entire log and removes overlapping information on the fly.

After test execution (Phase 2), Yardstick uses the coverage trace to compute the re-

Operation	Description
`PacketSet empty()`	Return an empty set of packets
`PacketSet negate(P)`	Return the set of packets not in the input set
`PacketSet union(P1, P2)`	Return the union of two packet sets
`PacketSet intersect(P1, P2)`	Return the intersection of two packet sets
`Boolean equal(P1, P2)`	Return if two packet sets are equal
`PacketSet fromRule(rule)`	Convert the match field of a rule to the corresponding set of packets
`Long count(P)`	Return the number of packets in the set

Figure 3.4: Operations over packet sets to help compute coverage.

quested metrics. This computation can be described using operations over packet sets shown in Figure 3.4. We compute coverage in three steps.

Step 1: Compute rule match sets. We first compute match sets of rules using the forwarding state. Our network and test models have disjoint match sets for rules in the same table. Given a device v and its forwarding rules $S[v]$, we compute the disjoint match set of each rule by walking the ordered list of device rules, computing the match set of each as the intersection of its match field and packets not matched so far [79].

Step 2: Compute covered sets. From the coverage trace (P_T, R_T), we compute the covered sets of all rules in the network using Algorithm 1. If a rule exists in R_T (reported by inspection tests), its covered set is its match set. Otherwise, it is the intersection of its match set and tested packets P_T.

Step 3: Compute coverage metrics. In the final step, Yardstick computes the component-level and aggregate coverage metrics, using the framework of §3.3.3. These computations are straightforward for all metrics except for path-based metrics. Unlike other metrics, information about neither the set of covered paths nor the number of all paths is readily available in the coverage trace. For boundedness (§3.2.2), we need the number of all paths as the denominator for aggregate metrics–if we see 10 paths in the coverage data, is that

Algorithm 1: Compute covered sets $T[r]$

Input: Network $N = (V, I, E, S)$, Test suite T

```
1 Procedure ComputeCoveredSets()
2     for r ∈ ⋃_{v∈V} S[v] do
3         if r in R_T then
4             T[r] ← M[r]
5         else
6             T[r] ← intersect(P_T, M[r])
7     return {T[r_i] | r_i ∈ ⋃_{v∈V} S[v]}
```

out of 10 possible paths (100% covered) or 1000 (1% covered)? This count may be known for some types of structured networks but not in the general case.

When the count of paths is not known apriori, we consider all possible paths imputed by the network forwarding state as the total number of paths. All possible paths cannot be computed based on topology alone because many unrealistic zigzag paths (*e.g.*, a 20-hop leaf-spine-leaf-spine$\cdots$ path) will inflate the path count. We thus use the network forwarding state to compute paths that carry non-zero traffic. Computing the path count in this manner has risks because network state bugs can change the count.[3] We can guard against this risk by flagging to the user when the size of path universe changes dramatically relative to prior state snapshots. Absent major operational changes to the network, this universe is not expected to change significantly from day-to-day.

We compute the path universe by symbolically exploring the journey of all possible headers from all starting locations. This traversal is depth-first on the topology and emits new paths incrementally (i.e., first a single-hop path with the source, then a two-hop path with the source and its first neighbor, assuming that the source sends a non-empty set of

[3]Path-based metrics in the software domain have the same risk.

packets to this neighbor, and so on). We do not store all paths in memory–there can be 100s of millions of paths in a large network—but process them on the fly.

Processing a path involves computing its coverage using the coverage trace per Equation (3.3). When a path is emitted, we know its sequence of rules. We do not know its guard set of packets. Strictly speaking, we do not need to know which packets are in the guard set; we only need to know the size of this set. In the common case, when any transformations along the path are only one-to-one, this size is the same as that of the final packet set at the end of our path exploration, and we use that size. In other cases, we compute the guard set by reversing the forwarding operations using the final set of packets. At each hop this computes the input set of packets that can produce the output set. We encode rule actions as BDDs, which allows for quickly doing such computations.

3.5 Implementation

Yardstick is implemented using 2300 lines of C# code, not counting the lines in various third-party libraries. The packet set operations in Figure 3.4 are implemented using binary decision diagrams (BDDs) that can efficiently encode and manipulate large header spaces.

We also instrumented a testing tool to report coverage to Yardstick. This tool is deployed in Microsoft Azure and supports all the test types mentioned in §3.2.1. Because the information Yardstick needs is readily available, we had to change only 7 lines of tool's code to report coverage. Each such line was an API call inserted at an appropriate place in the testing logic. Yardstick links to the tool as a dynamic library, and the two share type definitions for objects such as packet sets. Such an integration helps eliminate serialization-deserialization overhead. (Other tools integrated with Yardstick would likely incur higher data processing overhead.)

By default, Yardstick computes coverage for all device, interface, and rule coverage in the network. Users can customize the coverage report in a few different ways. They can request path coverage, which is not computed by default because of its high computational cost (§3.7). Users can also narrow the coverage analysis to particular flows by specifying

the flows' start locations and header spaces.

Finally, users can zoom in on a subset of components by providing a binary function that takes in the component and returns true if that component should be considered. This filtering option is helpful when users want to analyze coverage for, say, only leaf routers or only inside-to-outside paths. In the future, we plan to let users provide a coverage specification directly, allowing them to compute coverage for a richer set of component types.

3.6 Case Study: Azure Datacenter Networks

Yardstick is deployed in Azure as part of a service to evaluate the impact of changes to production networks. The service computes the network forwarding state that results from the change and then uses a test suite to check if the state is correct. We present a case study based on one month of Yardstick deployment in one of the networks. Its coverage reports identified systematic testing gaps in the original test suite and helped improve the test suite by suggesting new tests to address those gaps. These new tests are now part of the network's test suite.

3.6.1 Network Overview

Our case study focuses on a regional network that interconnects hundreds of thousands of hosts across multiple data centers in the same geographic region.

Topology and routing. Each datacenter network is a hierarchical Clos topology [33]. The top-of-rack (ToR) routers are at the bottom, and they directly connect to hosts. Aggregation routers connect the top-of-rack routers together in pods, which in turn are connected via a set of spine routers. At the top of each data center, the spine routers are connected to multiple layers of regional hub routers [49] which interconnect datacenters within a region. The regional hub routers are further connected to wide-area backbone routers that provide connectivity to the Internet and to other regions.

The network uses eBGP routing protocol on all routers [47]. Each router is assigned a private BGP ASN based on its role in the datacenter and configured with the `allow-as-in` command to avoid rejecting valid paths as loops [47] (*e.g.*, a ToR1-Aggregation1-ToR2 path is legal even though the two ToRs have the same ASN). As a fail safe, every router is also configured with a default static route (for the prefix `0.0.0.0/0`) that forwards packets to connected, higher-layer ("northern") neighbors. In the event of many kinds of failures, this backup measure ensures that packets will still have connectivity. For redundancy and load balancing, eBGP equal cost multipath (ECMP) is enabled on all routers.

Each ToR connects to hosts via Ethernet interfaces with assigned subnets, and, to enable connectivity to those hosts, it advertises aggregated prefixes for its directly connected hosts into the eBGP routing protocol. In addition, each router has one or more loopback interfaces whose corresponding connected routes are injected into eBGP via route redistribution.

Testing Pipeline. The network undergoes frequent changes in response to planned maintenance, migrations, and policy updates. All major changes are rigorously tested prior to deployment using a two-step process. An in-house simulator [50] and emulator [49] compute the network forwarding state that will result from the change. This state is then tested using a test suite defined by network engineers. Test execution produces a pass-fail report that network operators analyze to determine if the change is safe. Human oversight is needed here because it is possible that tests may fail as a result of modeling error or transient failures.

Yardstick has been integrated into this pipeline, where it augments the test results with coverage metrics. Its output is analyzed by a network engineer to improve the test suite and by network operators to determine the safety of the change.

3.6.2 *Identifying Testing Gaps*

Prior to the introduction of Yardstick, the network's testing pipeline used a collection of tests of two types:

1. *DefaultRouteCheck:* a subset of RCDC [42] contracts related to the default route that check that default routes have the correct set of next hops. This a state-inspection test.

2. *AggCanReachTorLoopback:* check that the aggregation routers correctly forward packets for ToR routers' loopback interfaces. This is a local symbolic test.

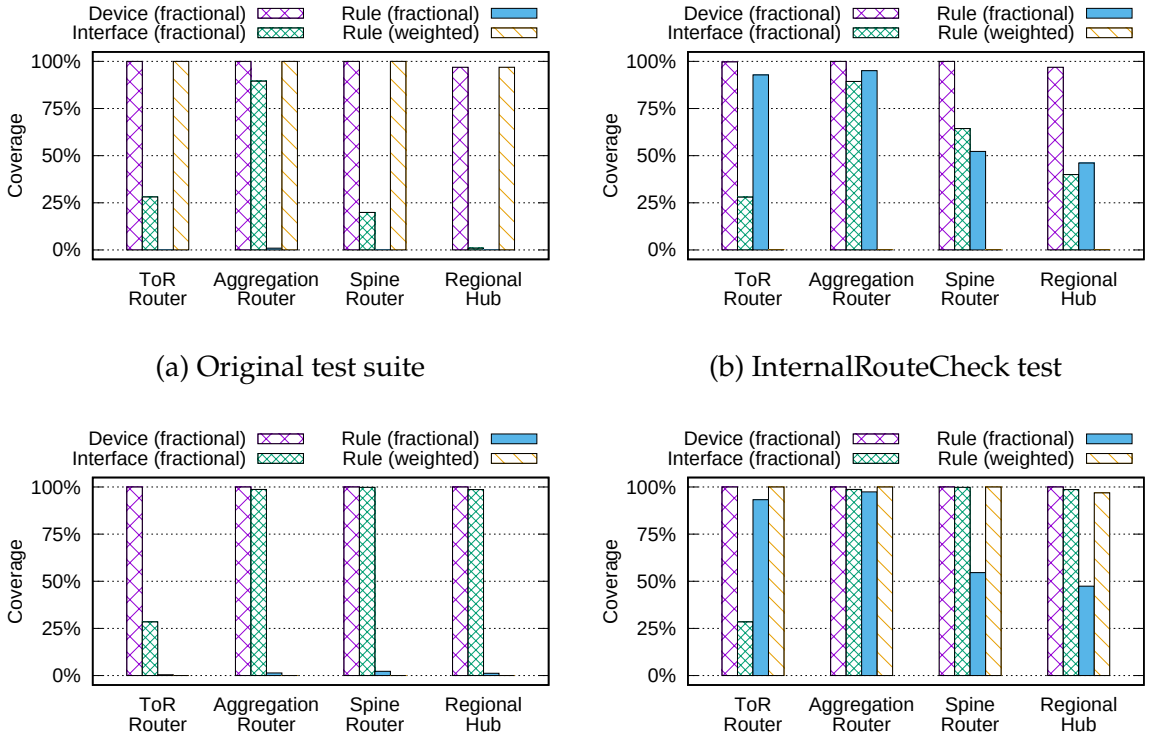
Figure 3.5a shows the coverage that Yardstick reported for this test suite. We break results by router type and plot fractional averages (§3.3.3) for devices, rules, and interfaces. We also plot the weighted average for rules; weighted average for devices and interfaces (not shown) was similar to that for rules. This view of the data was particularly useful toward understanding testing effectiveness and gaps for this network.

We see that fractional device coverage is close to perfect for all types of routers.[4] This high coverage is because the DefaultRouteCheck covers all devices. Device coverage is slightly low for regional hub routers because some of those routers are not expected to have the default route, and so the test excludes them.

Interface coverage, on the other hand, is quite uneven. It is high for aggregation routers because of the AggCanReachTorLoopback test but low for other router types. If the default route is not using the interface, it is not being tested at all. Thus, primarily northbound interfaces (*i.e.*, toward the higher layers in the hierarchy) are tested.

Rule coverage tells an interesting story and also shows the value of different types of aggregations. Fractional rule coverage is really low—most of the bars for this measure are indistinguishable from 0 in Figure 3.5a—indicating that the vast majority of the rules are

[4]Yardstick did actually reveal some completely untested devices in the network. This discovery was initially surprising to engineers. It later became clear that the untested devices were legacy routers that could not be tested. We have excluded such devices from our analysis.



(a) Original test suite

(b) InternalRouteCheck test

(c) ConnectedRouteCheck test

(d) Final test suite

Figure 3.5: Coverage for different tests.

untested. But weighted rule coverage, which weighs each rule by the size of the match set, is high because the default route matches the vast majority of the destination IP address space (anything that is not covered by a more specific rule). This data supports the engineers' intuition to prioritize default routes in their testing.

Low fractional rule coverage became the catalyst for identifying testing gaps. Focusing on forwarding rules that Yardstick reported as untested, we identified three categories of routes.

1. *Internal routes.* Each device has many prefixes to destinations that are internal to the region. These include prefixes for hosts connected to ToRs and prefixes of loopback

interfaces on all routers.

2. *Connected routes.* Each device has connected routes that correspond to its physical and aggregated interfaces, which have statically configured /31 (IPv4) and /126 (IPv6) prefixes. Because these prefixes are used for point-to-point connections only, they are not redistributed into the global eBGP routing protocol. Yardstick flagged that none of these connected routes, and many of their associated interfaces, were untested.

3. *Wide-area routes.* Yardstick revealed that routers in the upper layers of each data center had particularly low rule coverage. Upon further investigation, we found that these rules were for routes learned from the wide-area network, which are advertised to the regional hub and spine layers but are not leaked into lower layers.

At first blush, it may appear that a system like Yardstick is an overkill for identifying such gaps–engineers should have known about them based on their knowledge of the network and the test suite. But the real-world aspects of this challenge are worth noting. First, the engineers who originally developed the test suite can be different from those who are now responsible for maintaining and improving it. It can be difficult for the latter group to look at the old test code and judge what the tests are not covering. Second, real networks lose their original design simplicity and symmetry over time. This evolution makes it difficult to reason about network structure in one's head, to accurately identify which components are tested and which ones are not. The coverage information that Yardstick provides from various perspectives is thus key to comprehensively and reliably identifying testing gaps.

Yardstick also makes it easy to focus one's efforts on the most productive kind of test development—the creation new tests that provably improve coverage—rather than on development of redundant tests that do little to find additional errors in networks. We discuss the new tests developed for our network next.

3.6.3 *Toward a High-coverage Test Suite*

After identifying the testing gaps above, the engineers authored two new tests for the network.

1. *InternalRouteCheck*. This test validates that all prefixes that originate within the datacenter (*i.e.*, the internal destinations) are forwarded through and only through the full set of topological shortest paths. The design of the network is such that internal destinations are routed along shortest paths and there are many such paths. The test is implemented as a local symbolic test that uses RCDC's idea [42] of decomposing an end-to-end invariant into local forwarding contracts that dictate the next hops for a prefix at a device. Suppose a device v originates a set of prefixes $\{p_v\}$, which include host subnets and loopback addresses. To compute the local contracts for $\{p_v\}$, the test first perform a breadth-first search from v to compute shortest distances from all other devices. If the device v' is d-hops away from v, it should then forward $\{p_v\}$ to all its neighbors with distance of $d-1$ to v.

2. *ConnectedRouteCheck*. This test validates that both ends of a physical link have the connected route to the assigned /31 and /126 prefixes. It is a state-inspection test.

The engineers have yet to define a test for wide-area routes, the third gap that Yardstick helped identify. The challenge is that there is not yet any specification of the routes to expect from the wide-area network.

The two tests above are now deployed.[5] The coverage of these tests is shown in Figures 3.5b and 3.5c. InternalRouteCheck covers over 90% of the rules on ToR and aggregation routers, and around 50% on spine and regional hub routers. Its impact differs

[5]These tests did not identify unique bugs during the study. All discovered bugs were shallow and were flagged by the DefaultRouteCheck as well. However, one cannot rely on bugs being always shallow. Yardstick enabled the engineers to add tests that improve coverage, which improves the chances of finding additional bugs and increases confidence in the correctness of network changes.

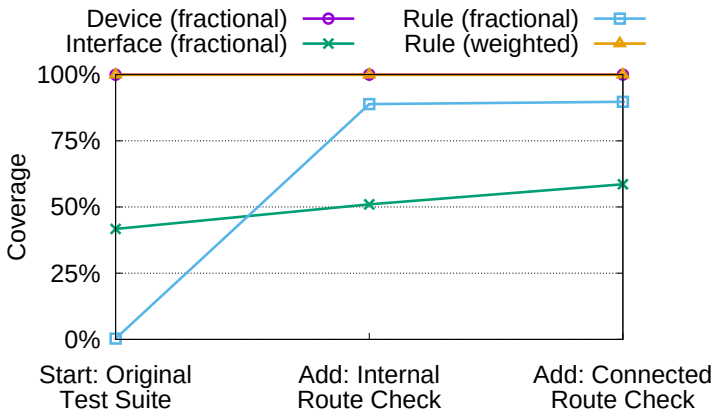


Figure 3.6: Coverage improvement with test suite iterations.

across router types based on the fraction of internal routes that a type contains. ConnectedRouteCheck covers nearly 100% of the interfaces on all routers except for ToRs.

The coverage for the final test suite, after adding these two tests to the original test suite, is shown in Figure 3.5d. The coverage is substantially higher than before. But fractional rule coverage on spine routers and regional hubs is only around 50% (because of untested wide area routes). Fractional interface coverage on ToR routers is only 25%. This coverage level is similar to that of the original test suite and of the two new tests individually, indicating that all tests are covering the same set of interfaces. We discovered that host-facing interfaces are not being tested, and as a result, will be developing another new test for these interfaces soon.

Figure 3.6 summarizes the coverage improvement across all devices during the course of our study. Within the first month of its deployment, Yardstick helped improve rule coverage by 89% and interface coverage by 17%.

3.7 Performance Evaluation

We conducted controlled experiments to benchmark the performance of Yardstick along two measures of interest: *i*) the overhead of tracking coverage when the tests are running;

and ii) the time it takes to compute the metrics after the tests are done. We generate synthetic fat-tree networks [2] of different sizes by varying the topology parameter k between 8 and 88, which generates networks of up to 9680 routers. Each ToR has one hosted prefix, and network routing functions as described in §3.6.1.

All experiments were performed on a desktop PC with an 8C8T Intel CPU running at 4.9 GHz and 16 GB of DRAM.

3.7.1 Overhead of coverage tracking

We measure the overhead of coverage tracking by running tests with and without tracking enabled. We consider four types of tests:

- *DefaultRouteCheck* is the state-inspection test mentioned earlier. It determines whether each switch has a default route that forwards to higher-layer neighbors.

- *ToRReachability* is an end-to-end symbolic test. It checks that all packets that originate at a ToR with a destination IP address in the hosted prefix of another ToR can reach the correct ToR.

- *ToRContract* is a local symbolic test. It checks the same invariants as ToRReachability, but does so by decomposing the invariant into a local forwarding contract for each router. ToRContract is a subset of RCDC [42].

- *ToRPingmesh* is an end-to-end concrete test. It checks the same invariants as as *ToRReachability*, but samples a single address from each prefix instead of reasoning about all packets. The idea of testing ToR pairs using concrete packets is drawn from Pingmesh [36].

Figure 3.7 shows the time to execute each test with and without coverage tracking enabled. We see that the overhead of coverage tracking is small. The worst case absolute time overhead is 54 seconds, which occurs for the ToRReachability test in the largest topology.

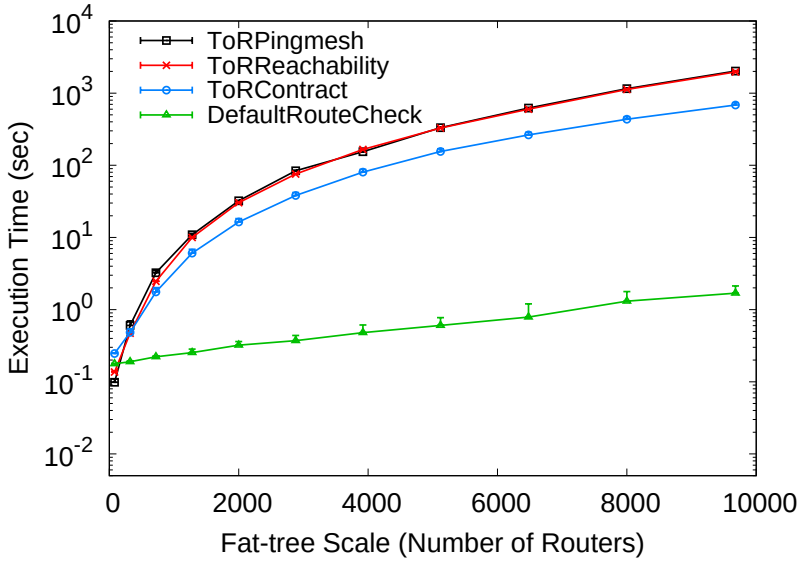


Figure 3.7: Overhead of coverage tracking. The lines show the test execution time without coverage tracking, and the error bars show the time with coverage tracking.

In this case, the baseline (coverage disabled) test time is 1967 seconds, and thus the relative overhead is only 2.8%.

The worst case relative overhead is 52%, which occurs for DefaultRouteCheck in the topology with 6480 nodes. In this case, the baseline test time is only 0.79 seconds. (State-inspection tests are lightweight.) Across all cases where the baseline test takes over a minute, the relative overhead of tracking coverage is under 10%.

3.7.2 Performance of coverage computation

After the tests finish, Yardstick computes coverage metrics using the coverage trace and network state. Figure 3.8 shows the computation time for different components (when computed by itself) as a function of network size. The coverage trace for this experiment is from the tests in the previous section. We show results for fractional averages; performance is similar for other aggregations.

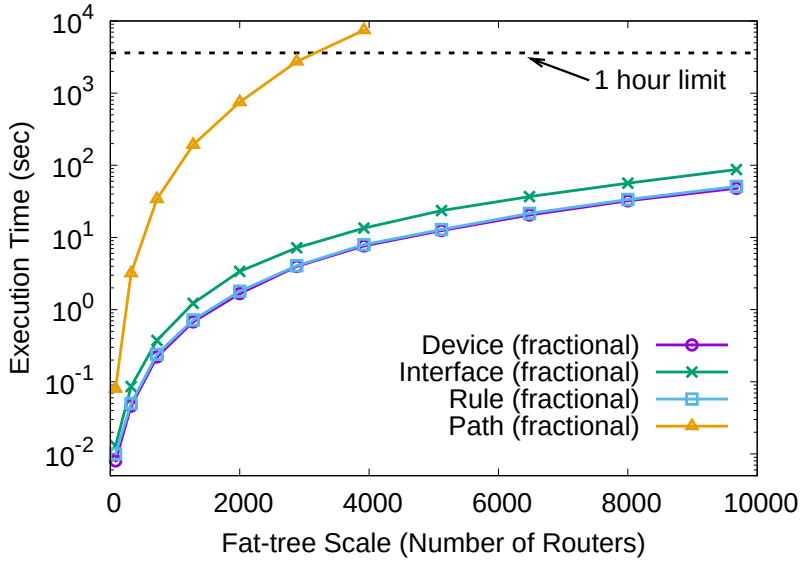


Figure 3.8: Time to compute coverage metrics.

We see that local metrics (*i.e.*, device, interface, and rule coverage) are reasonably fast to compute. Each metric is computed in less than 90 seconds, even on the largest topology. We also find that because a fair bit of processing is shared among these metrics (*e.g.*, computing match sets of rules), computing all of them together takes only 91 seconds (not shown in the graph).

Path coverage, on the other hand, is computationally expensive. It takes 45 minutes on the 2880-node network. On larger networks, its execution time exceeds the 1-hour timeout we used for these experiments. This metric is expensive because it requires iterating over all possible paths in the topology. As our networks use multi-path routing, doing so becomes prohibitive beyond a certain point.

Network engineers that we have talked to mentioned that Yardstick is efficient enough to be useful in practice. Coverage metrics are not expected to change significantly at short time scales unless the network state or the test suite changes significantly. The engineers thus mentioned computing path-based expensive metrics once a day, while relying on the local metrics to more quickly catch regressions in testing.

3.8 Related Work

Our work builds on two distinct lines of research. The first line of research involves network testing and verification tools [36, 42, 45, 28, 80, 46, 38, 79]. These tools have developed a range of techniques for scalable analysis of network data and control planes. We have borrowed some of these ideas (*e.g.*, the use of BDDs) to compute coverage metrics efficiently. However, the goals of our research are different from, and complementary to, this past work: we have developed metrics that systematically quantify how well verification and testing tools have been put to use by network engineers in practical industrial settings.

Second, we borrow from the software domain the idea of using coverage metrics to quantify test suite quality and reveal testing gaps. There, many metrics have been developed over the years to help software engineers [39, 7, 32]. Our network coverage metrics are specialized for operation over network forwarding state.

We share with ATPG [81] the goal of improving network testing. ATPG crafts test packets that exercise each rule in the network forwarding state to validate that routers indeed forward the packet as per their state. It thus improves testing that aims to find bugs in device software and hardware responsible for forwarding packets. In contrast, we improve testing that aims to find bugs in the forwarding state itself. In service of this goal, we also develop several notions of coverage beyond rule coverage.

We introduced the idea of using coverage metrics to improve the use of network verification in a position paper [14]. That paper did not develop a computational basis or a system to compute coverage metrics, nor did it report on experience of using coverage metrics to improve test infrastructure of large-scale, industrial networks.

3.9 Conclusion

We described a framework to define and compute network coverage metrics in order to help engineers assess and improve the quality of their test suites. To be able to compute a

range of metrics using a diverse set of tests, we must decompose both the state exercised by testing and network components into atomic testable units. We built our system Yardstick to perform these tasks and deployed it in Microsoft Azure. Within the first month of deployment in a production networks, Yardstick expanded testing to 17% more network interfaces and 89% more rules.

These results notwithstanding, we believe that we have barely scratched the surface of network coverage metrics. For software, the development of coverage metrics and usable systems has been a multi-decade journey that still continues. We expect networking to take a similar journey, with researchers developing increasingly sophisticated metrics for a range of settings. The framework developed in this paper can provide a useful starting point for that journey.

Chapter 4

NETWORK CONFIGURATION COVERAGE

The previous chapter have proposed data plane coverage to reveal testing gaps. It shows which data plane elements, such as forwarding rules, are exercised by a test suite. However, well-tested data plane does not imply well-tested configurations. Data plane elements are the output of network's configurations and the current operating environment (failures, external routing information). Testing a given data plane only tests configuration elements that are exercised in that particular environment. Other configuration elements are not tested. We demonstrate this empirically via a scenario in §4.6 where testing *all* data plane elements leaves over half of configuration lines untested.

In this chapter, we develop *configuration coverage* to provide comprehensive and precise feedback to network engineers on test suite quality. Our goal is to identify exactly which configuration lines are tested and which ones are not. We want to consider all configuration elements, not only those that contribute to the current data plane. Revealing exactly which lines are untested helps improve tests—add tests that target untested lines—which in turn can improve network reliability. This is similar to how code coverage tools help improve tests and software reliability [41, 19, 24].

4.1 Defining Configuration Coverage

We deem a configuration element to be covered if it i) is tested directly by a control plane test; or ii) contributes to the production of a data plane state element (i.e., an entry in the protocol or main RIB) tested by a data plane test. For now, assume that contributions are deterministic. We discuss non-deterministic contributions in the next section.

Figure 4.1 illustrates configuration coverage as a result of a data plane test. It shows

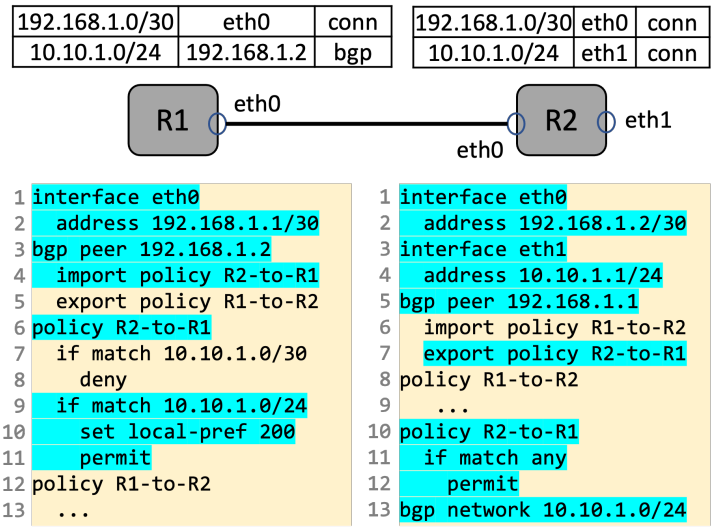


Figure 4.1: An example network with routing tables and configurations. The highlighted configuration lines are covered when the route to 10.10.1.0/24 is tested at R1.

parts of the two routers' configuration. R1's configuration defines one interface (Lines 1-2) and one BGP peer (192.168.1.2, which is R2's address), and it specifies the import and export policy to use. The import policy (R2-to-R1 at Lines 6-11) denies routing messages for a particular prefix and sets the preference for another.

R2's configuration defines two interfaces, a BGP peer (R1) and routing policies. At Line 13, it states that the prefix 10.10.1.0/24 should be announced to BGP peers *iff* it is in the main RIB.[1] In our example, 10.10.1.0/24 will be in the main RIB as it corresponds to the eth1's prefix. (Address statements like Line 4 encode the IP address and prefix length. For eth1, given the address 10.10.1.1 and prefix length of 24, the prefix is 10.10.1.0/24.) Routers add interface prefixes to the "connected" protocol RIB, from where those prefixes can enter the main RIB. The resulting RIBs on the two routers are shown in the figure. Each entry includes the next hop and source routing protocol ("conn" = connected).

[1]Different router vendors have different semantics for BGP network statements. We are assuming Cisco semantics.

Suppose the entry for 10.10.1.0/24 at R1 was tested by a data plane test. The covered configuration elements are highlighted. On R1, the BGP peer configuration and import policy binding (Lines 3-4) are covered because the tested entry came via that peering and passed through that policy. Parts of the routing policy R2-to-R1 relevant to the tested state (Lines 6, 9-11) are also covered. The interface definition (Lines 1-2) is covered because it enables the BGP peering to be established. In contrast, the export policy R1-to-R2 and unexercised parts of R2-to-R1 (Lines 7-8) are not covered.

There are covered configuration elements at R2 as well. These include the interface definitions—eth0 enables the BGP edge and 10.10.1.0/24 was announced due to eth1—and BGP peering, the export policy, and the BGP network statement.

Alternative definitions of coverage. One may consider an alternative definition of coverage that disregards non-local configuration elements. But we posit that including non-local elements is more meaningful. These elements, such as the BGP network statement on R2's Line 13, are just as key to the existence of 10.10.1.0/24 at R1 as the local elements.

Another definition of coverage is based on mutation [7]: a configuration line is deemed covered if its mutation alters the test result. Compared to the definition of coverage we adopt, mutation-based coverage will report an additional class of configuration elements as covered—configuration elements that de-prioritize (or reject) the competitors of the tested data plane element. Mutation-based coverage tends to be significantly harder to compute [43], and its results can be hard to interpret. In developing the first tool in this space, we decided to focus on a simpler, more direct definition of coverage. We will explore more sophisticated definitions in the future.

4.1.1 Our approach

While it is straightforward to identify configuration elements covered by a control plane test, it is not so for data plane tests. Data plane tests analyze the "output" of the control plane, and we need a scalable way to compute which configuration elements contributed

to tested data plane state. The relationship between these inputs and outputs is complex. How a particular RIB entry comes about relies on many configuration elements across multiple devices. The need to map tested outputs to input space sets computation of configuration coverage apart from data plane coverage and software coverage, for both of which the coverage domain is the same as test domain.

To motivate our approach to solving this problem, let us first sketch two strawman approaches. One potential approach is to express control plane computation declaratively, e.g., in Datalog. This enables identification of contributing inputs for a given output using a form of backward-reasoning [84, 77]. However, network control plane computations can be quite complex (e.g., non-monotonic behaviors [34, 60]). While declarative encodings may work in special cases [50], it is generally hard to get high-fidelty, performant encodings. That is why most control plane analysis tools use an imperative approach [80, 28, 54, 55].[2]

Another potential approach is to use simulation-based forward reasoning, i.e., simulate the control plane (imperatively) and track which configuration elements feed into each part of the data plane state. However, this approach has scalability limitations. Network simulation is time and memory intensive [28, 80, 55], and it will become significantly worse if it needed to track all necessary information along each hop.

Our approach is based on two observations. First, for the purposes of computing coverage, we do not need a full computational model of the control plane. We need to only track which configuration elements contribute to tested data plane state (i.e., taint analysis [72]), not the exact input-output relationship; and we need to reason only about the stable state (i.e., the the of devices once they have settled on best paths), not the transient states. Data plane testing [45, 46, 38, 79, 42] assumes that the analyzed state is stable. Our second observation is that the stable state contains enough information for us to infer contributions of configuration elements after the fact, based on the semantics of the control

[2]Batfish [28], a widely used control plane analysis tool, originally used Datalog to encode network control planes but switched to imperative simulations due to expressiveness and performance challenges.

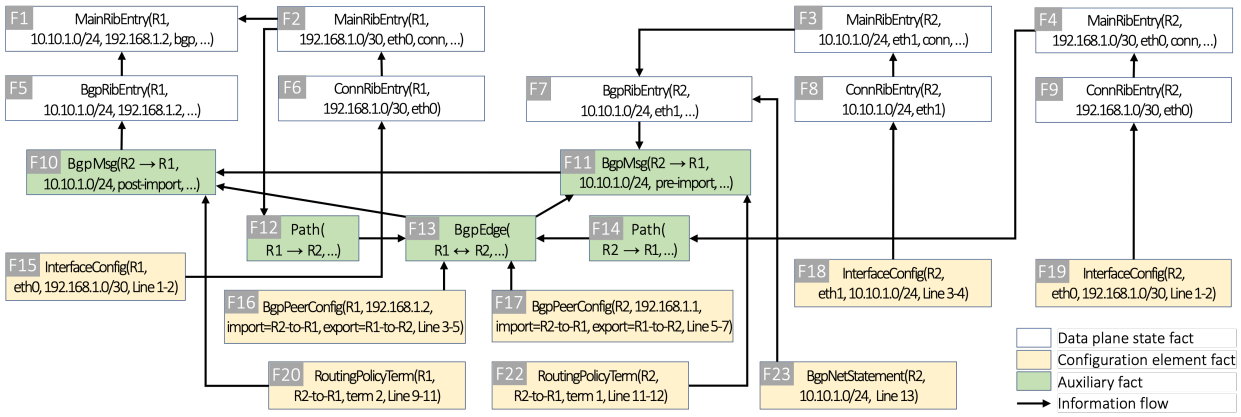


Figure 4.2: Subset of the IFG for the Figure 4.1 example. It tracks configuration elements that contribute to the tested RIB entry (F1).

plane. This inference is vastly cheaper than tracking contributions towards all data plane state entries, independent of whether they are tested.

To model contributions to the stable state, we use an *information flow graph* (IFG). Figure 4.2 shows a subset of the IFG for the example in Figure 4.1. Each node is a *fact* and arrows denote information flow from the tail to head. IFGs have three types of facts: i) data plane state, ii) configuration elements, and iii) auxiliary facts that capture intermediate dependencies between data plane state and configuration elements.

The main RIB entry 10.10.1.0/24 at R1 (F1) is derived from the corresponding BGP RIB entry (F5), which in turn is derived from the BGP message from R2 (F10). This message exists because of the BGP edge between R1 and R2 (F13), the source message sent by R2 (F11), and the relevant configuration element within import policy (F20). R2 sent the BGP message because of the same BGP edge (F13), its export policy elements (F22), and the BGP RIB entry (F7). This BGP RIB entry exists because of the configuration element (F23) and the RIB entry (F3), which exists because of the connected route (F8). The BGP edge (F13) exists because of the configuration elements that define the peering (F16, F17) and paths between R2 and R1 that enable the BGP session to be established. The paths depend

on the RIB entries (F2 and F4, respectively), the contributions to which can be similarly traced. In this manner, the IFG captures all configuration elements that led to the tested RIB entry (F1).

We do not track IFG dependencies proactively but infer them on-demand based on control plane semantics, using a mix of backward-forward reasoning. Backward inference infers the parent (tail) of the edge from its child (head). The information in child nodes is not enough to fully recover the parent nodes, but is often enough to select them from the known stable state. For instance, we can compute the BGP RIB entry F5 from the main RIB entry F1—the main RIB entry indicates that its source routing protocol is BGP, and we thus look up the BGP RIB for 10.10.1.0/24.

Lookup-based inference does not always work. For instance, given a BGP message which has passed through an import policy, we cannot compute backwards which terms of the import policy were exercised (F10 $\leftarrow$ F20). Another parent of F10, the pre-import BGP message (F11) cannot be looked up either because it is not part of the input and needs to be computed on-the-fly. To address these limitations, we combine backward and forward inference. When a parent can not be directly looked up, we first look up the prerequisites of the parent. For instance, we can look up F7 based on F10. Next, we use targeted simulations to compute non-existing facts and to select relevant facts exercised in a control plane process or data plane process. For instance, given the BGP route at R2 (F7), we simulate its processing through the export policy, which allows us to derive the pre-import BGP message (F11) and find the policy term exercised during the export process (F22). Once F11 is computed, we conduct another targeted simulation to discover the policy term exercised in the import process (F20). Unlike a full control plane simulation, these targeted simulations are fast. They have limited scope (e.g., best path selection is not simulated) and are done only for messages of interest, not all messages.

By combining backward and forward inference, atop the stable state IFG, we can scalably discover all covered configuration elements. We describe this approach in detail next.

4.2 Design of NetCov

NetCov takes as input configuration files, data plane state (protocol RIBs, main RIB and active routing edges) of the network. The data plane state may be pulled from live network or produced by control plane analysis tools [55, 28, 80]. In addition, NetCov takes as input what is tested: data plane entries that are tested by data plane tests, and configuration elements that are tested by control plane tests. This information is produced by network testing tools [78, 28].

Based on these inputs, NetCov computes which configuration elements are covered. The core of this computation efficiently mapping a data plane fact to configuration elements that contribute to it. We describe this computation next.

4.2.1 *Information flow model*

IFGs are directed acyclic graphs whose nodes denote network *facts* and edges denote information flow between facts. Table 4.1 shows the types of network facts modeled by NetCov and the information flow between different types. Data plane state has three subtypes: main RIB entries, protocol RIB entries, and access control list (ACL) entries.

Auxiliary facts have three subtypes: routing edges, routing messages, and paths of routing messages. These facts are not strictly necessary, but they help create a compact IFG and speed up graph walking. For instance, the routing messages of many protocol RIB entries depend on the same path which in turn may depend on many main RIB entries. Adding an explicit fact for the path avoids the need to add all pairs of edges between routing messages and main RIB entries.

In our model, the auxiliary facts for routing messages represent messages between routing protocol instances across devices as well as within a device, i.e., redistribution [23]. This uniform treatment is a modeling convenience. In reality, explicit messages are not exchanged during redistribution (though redistribution is subject to routing policies akin to messages between cross-device routing instances).

Network fact		**Information flow**
Configuration element (c)		None
Data plane state	Main RIB entry (f)	$f_i \leftarrow r_j$ $f_i \leftarrow r_j, f_k$
	Protocol RIB entry (r)	$r_i \leftarrow m_j$ $r_i \leftarrow c_j$ $r_i \leftarrow f_j, c_k$ $r_i \leftarrow \{r_{j_1}, ...\}, c_k$
	ACL entry (a)	$a_i \leftarrow \{c_{i_1}, ...\}$
Auxiliary	Routing message (m)	$m_i \leftarrow r_j, e_k, \{c_{l_1}, ...\}$ $m_i \leftarrow m_j, e_k, \{c_{l_1}, ...\}$
	Routing edge (e)	$e_i \leftarrow \{c_{j_1}, ...\}$ $e_i \leftarrow \{c_{j_1}, ...\}, \{p_{k_1}, ...\}$
	Path (p)	$p_i \leftarrow \{f_{j_1}, ...\}, \{a_{k_1}, ...\}$

Table 4.1: Information flow model: Types of facts and all possible information flows for each type. $\{t, ...\}$ denotes a set of facts.

The last column of Table 4.1 shows how information flows among different types of facts. A main RIB entry stems from a protocol RIB entry and optionally another main RIB entry (when its next hop is an IP address whose corresponding output interface needs further resolution). A protocol RIB fact stems from a routing message (for protocols such as BGP), a configuration element (for connected interfaces and static routes), a main RIB entry accompanied with a configuration element (such as when a BGP network statement populates a main RIB entry into BGP RIB) or a set of RIB entries accompanied with a con-

figuration element (for aggregate routes). ACLs facts stem from configuration facts and have no other dependencies. Routing messages stem from a RIB fact or another message (e.g., post-import-policy message depends on pre-import-policy message), and they also depend on routing edges and routing policy configurations. Inter-device routing edges stem from paths that enable sessions to be established and configuration facts that define peerings; Intra-device routing edges stem from configuration facts that define redistribution. Finally, path facts depends on main RIB facts and ACL facts that impact routing traffic along the way.

For correct computation of coverage, the IFG model must be sound and realizable. Soundness means that it includes all relevant dependencies (per control plane semantics) and no more. Realizable means parents (tails) along all information flow edges can always be inferred, via lookup or simulation or a mix. Our model is sound to our knowledge; and that we are able to use it to compute coverage, using the framework described next, points to its realizability.

4.2.2 Inferring the IFG on demand

Based on the information flow model, NetCov uses a backward-forward inference framework to lazily materialize the IFG from any set of facts whose coverage need to be tracked. The framework is abstracted using a set of *inference rules* and an iterative construction algorithm. Each inference rule is function that takes a materialized IFG node as input and materializes a set of its ancestor nodes as well as the edges the allows the ancestors to reach the input node. These nodes and edges will be merged into the materialized IFG by the construction algorithm. The implementation of these functions uses one or both of the *lookup-based inference* and *simulation-based inference*. Let us elaborate.

Lookup-based inference. The computation of data plane state is lossy. While a main RIB entry may be derived from a BGP RIB entry, we cannot infer the complete BGP RIB entry from the main RIB entry because BGP specific attributes (e.g., AS-path) are not preserved

Algorithm 2: Rule to infer BGP RIB entry from main RIB entry.

```
1  def infer_from_main_rib_entry(f, stable_state):
2    if not (f is MainRibEntry and f.protocol == 'bgp'):
3      return []
4    bgp_entry = stable_state.bgp_rib.lookup(
5      host=f.host,
6      prefix=f.prefix,
7      nexthop=f.nexthop,
8      status='BEST'
9    )
10   return [(bgp_entry, f)]
```

in the main RIB.

To handle this information loss, our inference takes two steps. It first infers attributes that can be known from heuristics (we know such heuristics from control plane semantic, *e.g.*, the BGP RIB entry should have the same prefix as the main RIB entry derived from it). Next, we look up all entries in the stable state that match the inferred attributes. For instance, Algorithm 2 shows the simplified function to infer the BGP RIB entry that led to a main RIB entry. Based on control plane semantics, if a main RIB entry indicates its source protocol to be BGP, it must have stemmed from a BGP RIB entry on the same router with the same `prefix` and `nexthop` attributes (Lines 5-7). Besides, the BGP RIB entry should have been selected as the best route (Line 8). Such information is enough to uniquely identify the parent within the known stable state. The return value (Line 10) is a list of tuples denoting the IFG edges materialized by this rule.

Simulation-based inference. Lookup-based inference falls short in two scenarios. First, when a parent fact is absent from the known stable state (e.g., routing messages), and second, when the heuristics fail to infer enough information so as to uniquely identify the

parents (e.g., we cannot know which policy clauses are used in the production of a BGP route by looking at the resulted route). We use local simulations to complement lookup-based inference. But simulations can only be performed in the forward direction, *i.e.*, to compute a fact using simulations, we first need to know its parent. We use a generalized version of lookup-based inference to discover grandparent facts of a known fact, and then use simulations with the grandparents to infer their children (i.e., parents of the original fact).

Algorithm 3 shows the simplified inference rule that infers the ancestors of a post-import BGP message. Line 13 demonstrates the use of simulation-based forward inference to compute a missing parent fact on the fly. The two prerequisites to simulate the BGP message–the grandparent BGP RIB entry (`origin_entry`) and the BGP edge–are discovered via lookup-based backward inference, on Line 8 and Line 4 respectively. The simulation returns the derived BGP message after applying the routing policy, as well as the policy clauses exercised during the process. The second forward-simulation (Line 17) is to discover the policy clauses that are hit during the import process. The return value includes the inferred IFG edges that connect to the input node `m` as well as ones that connect to parent `pre_import_msg`. The former corresponds to information flow $m_i \leftarrow m_j, e_k, \{c_{l_1}, ...\}$ in Table 4.1 and the latter corresponds to $m_i \leftarrow r_j, e_k, \{c_{l_1}, ...\}$.

IFG construction. Next, we detail IFG materialization using inference rules. Assume for now that the information flow is deterministic; the next section discusses how we handle non-determinism.

As shown in Algorithm 4, the IFG initially contains only the nodes representing the tested data plane state facts from the input and does not have any edges (Line 2). It is then iteratively expanded by applying inference rules on existing nodes. In each iteration, all inference rules are applied to the dirty nodes derived from the previous iteration (Line 8). The new nodes and edges inferred during such process are collected and merged (with deduplication) into the IFG (Line 9-14). The computation repeats until no new facts can

Algorithm 3: Rule to infer ancestors of a post-import BGP message.

```
def infer_from_bgp_message(m, stable_state):
  if not (m is BgpMsg and m.is_post_import):
    return []
  bgp_edge = stable_state.bgp_edges.lookup(
    recv_host=m.host
    send_ip=m.nexthop
  )
  origin_entry = stable_state.bgp_rib.lookup(
    host=bgp_edge.send_host,
    prefix=r.prefix,
    status='BEST'
  )
  pre_import_msg, export_clauses = policy_simulation(
    input=origin_entry,
    policy=bgp_edge.export_policy
  )
  _, import_clauses = policy_simulation(
    input=pre_import_msg,
    policy=bgp_edge.import_policy
  )
  return [(pre_import_msg, m), (bgp_edge, m)] +
    [(cl, m) for cl in import_clauses] +
    [(origin_entry, pre_import_msg), (bgp_edge, pre_import_msg)] +
    [(cl, pre_import_msg) for cl in export_clauses]
```

be derived in an iteration.

Algorithm 4: IFG lazy materialization

Input: Initial nodes $\{v_i\}$; Inference rules $\{\phi_i : v \mapsto \{(u_i, v_i)\}\}$;

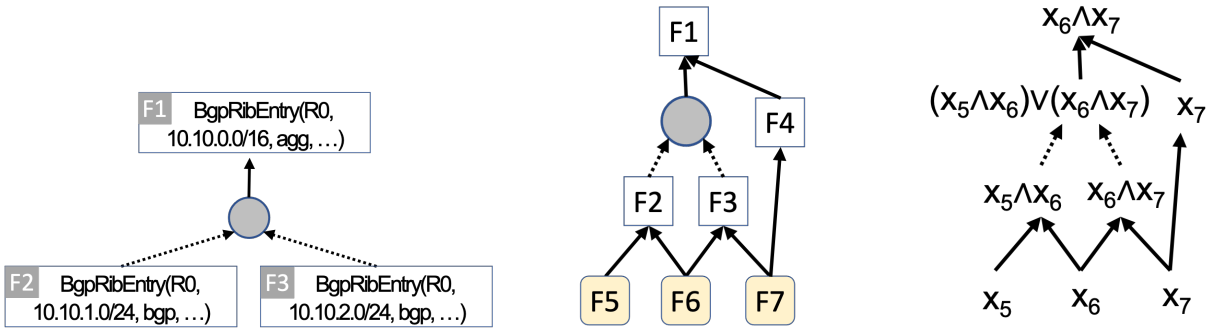
Output: Materialized IFG (V, E)

Data: Stable state data plane state (main RIB and protocol RIBs); Routing edges; Configuration elements;

```
1  Procedure BuildIFG({v_i}, {φ_i})
2      V, E ← {v_i}, ∅
3      V_prev ← {v_i}                          // dirty nodes of previous iteration
4      while |V_prev| > 0 do
5          V_curr ← ∅                           // dirty nodes of current iteration
6          foreach c ∈ V_prev do
7              foreach φ ∈ {φ_i} do
8                  E' ← φ(c)
9                  foreach (u_i, v_i) ∈ E' do
10                     if u_i ∉ V then
11                         V ← V ∪ {u_i}, V_curr ← V_curr ∪ {u_i}
12                     if v_i ∉ V then
13                         V ← V ∪ {v_i}, V_curr ← V_curr ∪ {v_i}
14                     if (u_i, v_i) ∉ E then E ← E ∪ {(u_i, v_i)}
15         V_prev ← V_curr
16     return (V, E)
```

4.2.3 Handling uncertainty

There are situations where it is not certain which stable state facts contributes to a given fact. One such scenario is BGP aggregation, where a prefix (e.g., 10.10.0.0/16) is added to the RIB iff at least one more of its more specific prefixes (e.g., 10.10.1.0/24) is present. When multiple more specifics are present, we do not know which one triggered the aggregate. Another such scenario is when multiple paths are available for a routing edge to be established, which can happen when the network uses multipath routing. Here, we do



(a) (b) (c)

Figure 4.3: Modeling uncertainty. (a) BGP aggregate (F1) has two potential contributors. (b) F5 is weakly covered but F6 and F7 are strongly covered. (c) The predicates of IFG nodes.

not know which path is actually used by routing messages.

It is important to model and report such uncertainty because the notion of contribution is different. Unlike deterministic contribution, when the contribution is non-deterministic, one or more parent facts can disappear without impacting the outcome represented by the child. Our experiments have scenarios where 78% of the configuration lines have non-deterministic contribution, and the tested fact would not be impacted if any of them did not exist. Not separating such uncertain contribution would lead to misplaced confidence in how well configurations are tested.

We model contribution uncertainty using *disjunctive* nodes in the IFG. This node points to the parent fact (e.g., the aggregated RIB fact) and the multiple contributors to the parent point to this node. See Figure 4.3a for an example where a BGP aggregate could be triggered by either of the two more specific prefixes. When our inference rules encounter uncertainty during IFG materialization, they produce a disjunctive node and attach all contributors to it as children.

We introduce the notion of *weak* coverage to capture the configuration elements whose

contribution to the tested facts is not critical. We define a contribution as non-critical if the tested fact will not be affected by deleting the configuration element from the IFG. In Figure 4.3b, F5 is weakly covered when F1 is tested because F1 can be derived without any contribution from F5, via F2 and F6. On the other hand, F6 is strongly covered because, without it, neither F2 nor F3 can be derived and thus the disjunctive node cannot be derived. F7 is also strongly covered because it contributes to F4, which is essential to F1.

NetCov labels each covered configuration element as strong or weak after the materialization of the IFG. The label is determined as follows. We first assign a Boolean variable to each configuration element in the IFG. Next, we build a Boolean predicate of each IFG node on top of these variables. The predicate of a fact depends on the predicate of its ancestors in the IFG: A normal node depends on the conjunction of its immediate parents, and a disjunctive node depends on the disjunction of parents. Therefore the predicate of any IFG node is ultimately composed of the variables associated with configuration elements that lead to it, denoted as $\Gamma(v) = F(x_1, \ldots, x_n)$. Figure 4.3c shows the predicates of IFG nodes in Figure 4.3b. We represent these Boolean predicates using Binary Decision Diagrams (BDDs) [18] and build BDD predicates by traversing the IFG. By definition, a configuration fact (denoted as x_i) is strongly covered if and only if there exists a tested data plane state fact (denoted as v), v is reachable from x_i in the IFG, and x_i is a necessary condition of $\Gamma(v)$. Therefore, once the predicates are built, we test graph reachability and logical necessity between each pair of configuration facts and tested data plane facts. Necessity $\neg x_i \Rightarrow \neg\Gamma(v)$ is equivalent to unsatisfiability of $\neg x_i \wedge \Gamma(v)$. While (un)satisfiability is NP-Complete in general cases, we note that it is efficient in our case—it can be reduced to computing the cofactor $\Gamma(v)|_{x_i=0}$ and testing whether the cofactor is constant false, both of which are efficient using BDD operations.

We further reduce the size of BDD predicates by precluding configuration facts that can reach tested facts via a path with no disjunctive node, such as node F7 in Figure 4.3b. These configuration facts must be strongly covered so their necessity do not need to be

tested. Besides, their validity variables can be replaced with constant true when building BDD predicates, which will not affect the strong/weak classification of other configuration elements. We empirically find this heuristic to be effective in reducing the number of variables used for weak coverage computation.

4.2.4 Future Extensions

Our current model tracks the contribution of configuration elements to concrete data plane state entries. While this view aligns well with tools that perform data plane testing [81], data plane verification [52, 45], and control plane testing [28, 80], it is not applicable to control plane verification tools [13, 1] that reason about data plane symbolically (i.e., simultaneously reason about multiple data planes under different environments). Control plane verification tools turn configuration into an internal model that is used for validation. NetCov can be extended to these tools by tracking how configuration elements contribute to the model, akin to how compilers link program source information to its intermediate representations.

The current implementation of NetCov supports BGP, a path vector protocol, and static routes. Other protocols, including link state protocols (e.g., OSPF) and label switching protocols (e.g., MPLS) can be supported with appropriate extensions. Such extensions require defining protocol-specific configuration elements and data plane state facts (such as label information base entry for MPLS) as well as all new information flows.

4.3 Implementation

We implemented NetCov with 4,000 lines of Python code. A total of 18 lambdas (Python functions) encode the IFG inference rules. NetCov uses Batfish [12] to extract configuration elements from configuration files and to run targeted simulations, and it uses CUDD [61] for BDD operations.

NetCov supports several major router vendors supported by Batfish, including Arista,

Type	Purpose
Interface	Interface and its settings (e.g., addresses)
BGP peer	BGP peer settings (e.g., IP address, AS number)
BGP peer group	BGP peer settings inherited by one or more peers
Route policy clause	One clause in an export or import route policy
Prefix list	List of prefixes, used in route policy clauses
Community list	List of BGP communities for route policy clauses
AS-path list	List of AS-path expressions for route policy clauses

Table 4.2: Configuration elements analyzed by NetCov.

Cisco, and Juniper. It builds a vendor-neutral representation of configuration elements using vendor-specific information provided by Batfish. Table 4.2 lists the configuration elements that NetCov currently analyzes.

NetCov may not consider all components of a device's configuration. One category of such components is device management configuration (e.g., login settings), which does not impact data or control plane functionality. The second category is control plane components that are not currently modeled by NetCov. This includes IPv6 (which is not modeled by Batfish currently) and routing protocols other than BGP (e.g., OSPF). The presence of unconsidered components does not imply that NetCov cannot be used for that network. As we show in the next section, NetCov provides helpful coverage information for parts that are considered.

After constructing the IFG, which yields information on which configuration elements are covered, NetCov computes which lines are covered. NetCov leverages the Batfish parser to map configuration elements to line numbers. Each element typically spans multiple configuration lines, and when an element is covered, it deems all of those lines as covered.

```
6880      policy-statement SANITY-IN {
6881          /* Reject any BGP prefix if a private AS is in the path */
6882      >   term block-private-asn {…
6885          }
6886          /* Reject any BGP NLRI=Unicast prefix if a commercial ISP's AS is in the path */
6887      >   term block-commercial-asn {…
6891          }
6892      >   term block-nlr-transit {…
6895          }
6896          /* Reject BGP prefixes that should never appear in the routing table */
6897          term block-martians {
6898      >       from {…
6921            }
6922            then reject;
6923          }
6924          /* Reject BGP prefixes which Abilene originates */
6925      >   term block-internal {…
6930          }
6931      }
```

(a) Line-level coverage. Green background denotes covered lines, and red denotes uncovered lines. Some lines are collapsed for simplicity.

Current view: top level - configs
Test: internet2.initial-tests
Date: 2022-09-20 14:54:06

	Hit	Total	Coverage
Lines:	16912	64886	26.1 %

Filename	Line Coverage		
atla.conf		24.1 %	1211 / 5019
chic.conf		40.5 %	4376 / 10800
clev.conf		32.9 %	1156 / 3512
hous.conf		24.9 %	1196 / 4801
kans.conf		20.0 %	1235 / 6178
losa.conf		20.4 %	1832 / 8960
newy32aoa.conf		11.8 %	770 / 6545
salt.conf		18.5 %	568 / 3063
seat.conf		30.6 %	1845 / 6030
wash.conf		27.3 %	2723 / 9978

(b) File-level aggregate coverage. The overall coverage is at top right, and the coverage for individual files (devices) is in the table.

Figure 4.4: Example `netcov` outputs.

Based on element and line coverage, NetCov produces three main outputs. The first is a coverage report at the granularity of individual lines (or elements). We produce this report in the `lcov` format, which is supported by common code coverage tools and enables users to visualize coverage results as annotations on configuration files. See Figure 4.4a for an example. The second is coverage aggregated at the file level, generated with the help of GNU LCOV [35]. See Figure 4.4b for an example. The third output is coverage aggregated by the type of configuration element, which shows what fraction of elements of each type are covered.

These outputs help users uncover testing gaps and improve their test suites in different ways. The aggregate results help identify systematic gaps such as "router A is poorly covered" or "routing policy clauses are poorly covered." The line-level results help them zoom in to specific gaps and develop tests that target them. The case study in the next section demonstrates this test suite improvement process.

4.4 Case Studies

We present case studies of using NetCov on two disparate networks, one a wide-area backbone and another a datacenter. In each case, using realistic test suites, we show that NetCov provides insight into what is and is not covered and how these insights help improve the test suites.

4.4.1 Case Study I: The Internet2 backbone

Internet2 is a nation-wide network that connects over 60,000 US educational, research and government institutions. The routing design of Internet2 is typical of backbone networks. It has 10 BGP routers spread across the country. The routers are organized as a single autonomous system (AS), and they establish iBGP full mesh on top of internal reachability provided by the IS-IS protocol. The Internet2 routers connect to 279 external BGP peers, and heavily use route import and export policies. The import policy for an external

peer has multiple policy statements, some specific to the peer and some shared within the same peer group. Peer-specific policies tend to specify a list of allowed prefixes from this peer, and others are used for sanity checking, preference setting, etc. Export policies are similarly structured.

Internet2's configurations that we study have 96,672 lines (in Juniper's JunOS format) across all routers. Of these, NetCov's coverage computation considers 64,886 lines. The bulk of the unconsidered lines correspond to device management, IPv6, and IS-IS protocol.

We do not have the data plane state of Internet2, which is needed to run data plane tests. We approximate it using Route Views [73], a repository of BGP routes from over two hundreds ASes worldwide. This data helps approximate BGP messages that external peers of Internet2 send to it. Consider a peer with AS number X. If we find a prefix P in RouteViews with AS-path $[A, X, Y]$, we assume that the peer sends P to Internet2 with AS-path $[X, Y]$. The existence of AS-path $[A, X, Y]$ means that AS A must have a route to P with AS-path $[X, Y]$, which it announces to its neighbors. If we find multiple AS-paths for a prefix, we pick the one with fewest AS hops.

We use these BGP messages that each peer sends to Internet2 as inputs to simulate Internet2's control plane using Batfish. The data plane state produced by this simulation is a coarse approximation of the real version, but it suffices to meet our goals of running data plane tests and characterizing configuration coverage.

Test suite coverage

To study how NetCov analyzes coverage for realistic test suites, we use the test suite proposed in Bagpipe [75]. It has three tests to validate Internet2's BGP configuration.

- *BlockToExternal*: ensure that BGP routes with BTE community are not announced to any external (eBGP) peer.

- *NoMartian*: ensure that incoming BGP messages from external peers for prefixes in the private address space ("Martian") are rejected.

- *RoutePreference*: ensure that if multiple routes to the same prefix are accepted from multiple external neighbors, the selected route belongs to the most preferred neighbor. The neighbor's preference depends on commercial relationship [29]. *Customers* are most preferred, followed by *p*eers, and then *providers*[3].

We implemented these tests using Batfish. *BlockToExternal* and *NoMartian* are control plane tests. *BlockToExternal* evaluates all BGP export policies on a set of BGP routes carrying the BTE community and asserts that the result be rejection. We generate the test cases by sampling BGP routes from the data plane state and attaching the BTE community to them. *NoMartian* evaluates all BGP import policies on a set of BGP routes destined for Martian addresses and asserts that the results be rejection. *RoutePreference* is a data plane test. It focuses on destination prefixes available via multiple neighbors and asserts that their local preferences reflect commercial relationship. We use CAIDA data [51] to infer commercial relationship between Internet2 and its BGP neighbors.

After running this test suite on Internet2, we find that it covers only 26.1% of configuration lines across all devices. Only a tiny fraction of configuration lines (0.5%) are weakly covered, so we do not separate weak/strong coverage for this case study; we will do that in the next one.

To help understand what is and is not covered in more detail, NetCov enables network engineers to look at the data from multiple perspectives. Figure 4.4b shows per-device coverage. We see notable variation across devices, from 11.8% to 40.5%. As we show below, the test suite has systematic gaps, and the cross-device variation stems from different devices having different fractions of covered configuration elements.

Figure 4.5 shows the coverage broken down by the type of configuration elements. For simplicity, we create four buckets of element types, as shown in the legend. The bottom bar shows the fraction of reachable configuration lines in each bucket. The "Test Suite" bar

[3]As a not-for-profit network, Internet2 treats its member institutions as customers and other not-for-profit networks (such as ESNet) as peers. Internet2 does not have providers in its routing preference model.

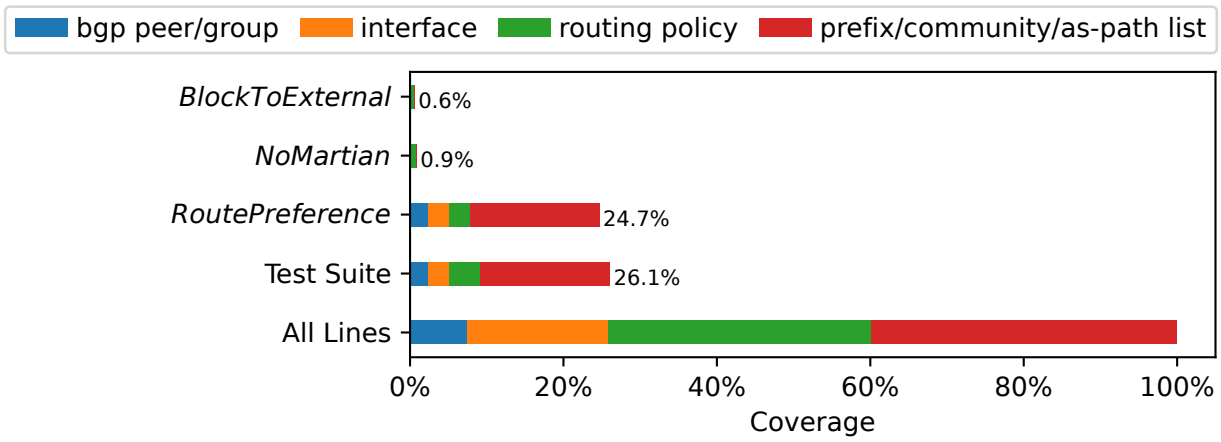


Figure 4.5: Coverage of the initial test suite broken down to each individual test and configuration type.

shows the covered fraction of those lines, and the top three bars show the coverage of individual tests. The total coverage of individual tests is 0.6%, 0.9% and 24.7% respectively. *BlockToExternal* and *NoMartian* cover only one type of configuration element (routing policies), and even within this type, they cover a small fraction. *RoutePreference* covered all four buckets but its overall coverage is still limited.

Finally, NetCov reports that 27.9% of configuration lines are "dead code" that will never be exercised. They include defined BGP peer groups with no members and defined routing policies that are never used for any peer.[4]

With 69% of BGP configurations, 85% of interfaces, 88% of routing policies, and 57% of route attribute match lists being completely untested, this test suite is clearly undertesting the network. This leaves the network vulnerable to bugs in untested configurations elements. Prior to NetCov, it was not possible for network engineers to get any insight into the quality of their test suite. It was also not possible for them to get help toward

[4]Per best practices, these lines should be deleted. Or, at a minimum, they should be tested lest someone start using an unused, erroneous policy. When it comes to testing, such lines can never be exercised by data plane tests, though control plane tests may be written for them.

systematically improving tests. We demonstrate this test suite improvement process next.

Coverage-guided test development

NetCov's feedback enables a test suite development process that enables users to systematically improve coverage, which helps test more critical aspects of the network and prevent outages. This process is iterative. In each iteration the user first identifies specific testing gaps and then creates new tests to target those gaps. We demonstrate the process using three iterations that focus on different types of gaps.

Iteration 1. We saw that routing policy coverage of *NoMartian* test is low (Figure 4.5) despite that it checks the import policies for all external peers. To investigate, we look at the structure of Internet2 import policies and find that routers have a policy named `SANITY-IN` which is shared by the majority of external neighbors. Figure 4.4a shows this policy with annotated coverage. Each router has an independent copy of this policy, but the copies and the coverage results are identical across routers. Of the five clauses in the policy, the clause `block-martians` starting at line 6,896 is the only clause that is covered. This coverage result confirms that the *NoMartian* test did its job, and more importantly, it revealed a systematic testing gap–the other four classes of forbidden routes are not being tested.

Once we know the gap, the solution suggests itself. We added a new test, *SanityIn*, to enforce that the other four classes of received BGP messages should be rejected. After adding this test, we used NetCov to confirm that this testing gap had been addressed. Routing policy coverage was improved by 0.6% and all five terms of `SANITY-IN` were covered by the new test suite. The quantitative improvement is low because `SANITY-IN` is just one of many policies in the network. With feedback from NetCov, network engineers can identify testing gaps in other routing policies and add more tests in a similar way.[5]

[5]Automatic test generation based on coverage feedback will further help engineers. We will investigate this in the future.

Iteration 2. BGP peer configuration coverage of *RoutePreference* test in Figure 4.5 is surprisingly low, given that all external BGP peers are supposed to be checked. Upon further investigation we find that the uncovered peers have permitted prefix-lists that do not overlap with other peers' lists, which left these peers untested.

We added a new test, *PeerSpecificRoute*, to check that BGP announcements received from external peers should be accepted if their prefixes is in a peer-specific prefix list. This test improved BGP peer coverage from 32% to 46%. The rest of untested BGP peers are either not allowed to send BGP routes to Internet2 or is intended for other internal use, such as monitoring and management. This test also improved prefix-list coverage from 45% to 63%. The remaining of untested prefix-lists are mostly (30% out of 37%) ones that are defined by never referenced.

Iteration 3. The low coverage of interface configuration in Figure 4.5 reveals another testing gap. *RoutePreference* is the only test in the initial test suite that checks interface configurations, and it only considers one category of interfaces–ones that are used to establish the tested BGP edges. Many other interfaces remain untested, including but not limited to ones that associate with untested BGP edges and other routing protocols, and the ones that are unused.

We added a new PingMesh-style [36] test, *InterfaceReachablility*, to check that the IPv4 addresses assigned to interfaces should be reachable from each router in the network. This test increased interface coverage from 15% to 53%. The rest of untested interfaces do not have IPv4 addresses assigned.

Figure 4.6 summarizes the coverage improvement for the three iterations of test improvement in our study. After only three iterations, the overall coverage was improved from 26% to 43%. This final coverage number is far from perfect, but our goal was not to develop the ideal test suite for Internet2; we wanted to demonstrate how coverage information helps develop new tests. Networks are complex, and we should not expect to get the job done with 6 tests. Many more tests are likely needed. With NetCov, network

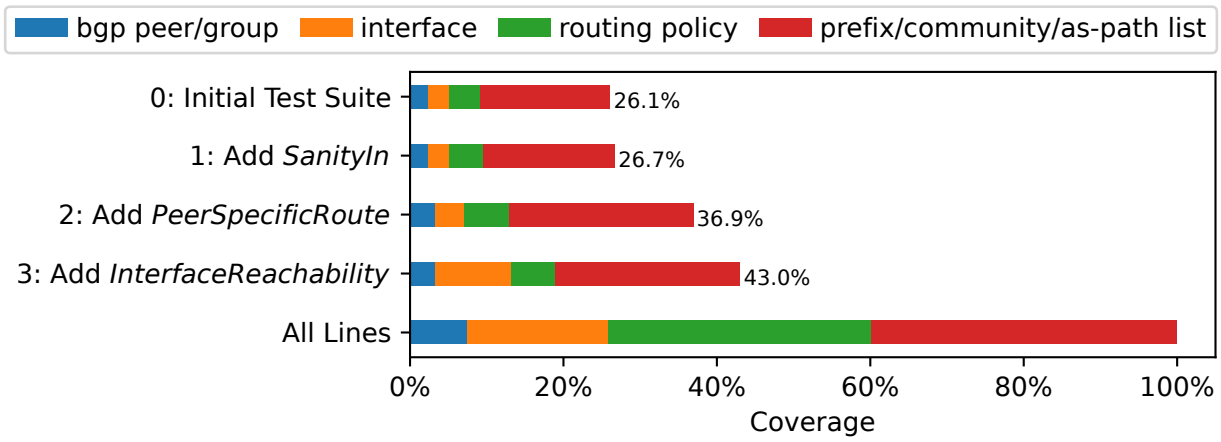


Figure 4.6: Coverage improvement with test suite iterations.

engineers now have a tool to develop new tests that meaningfully improve coverage.

4.4.2 Case study II: Datacenter networks

We study the coverage for data center networks which have a different topology and routing design. We create synthetic fat-tree[2] networks with routers across three tiers. The leaf routers at the bottom tier connect to hosts. Aggregation routers at the middle tier connect to leaf routers in a pod and to spine routers at the top tier. The spine routers connect to the wide area network (WAN). The WAN is not part of the tested network. Each leaf router is assigned a /24 prefix which is advertised inside the data center through eBGP. Spine routers receive a default route (prefix 0.0.0.0/0) from WAN via eBGP and propagate it to lower tiers. At each spine router, the entire address space of the network is summarized into a /8 prefix and is announced to WAN. Multipath routing (ECMP) is enabled with maximum number of paths set to 4. Routing policies are only configured at spine routers to white-list the default route received from WAN peers. We synthesize the configurations of these networks in Cisco IOS format.

We study a test suite of three tests inspired in prior works on data center network

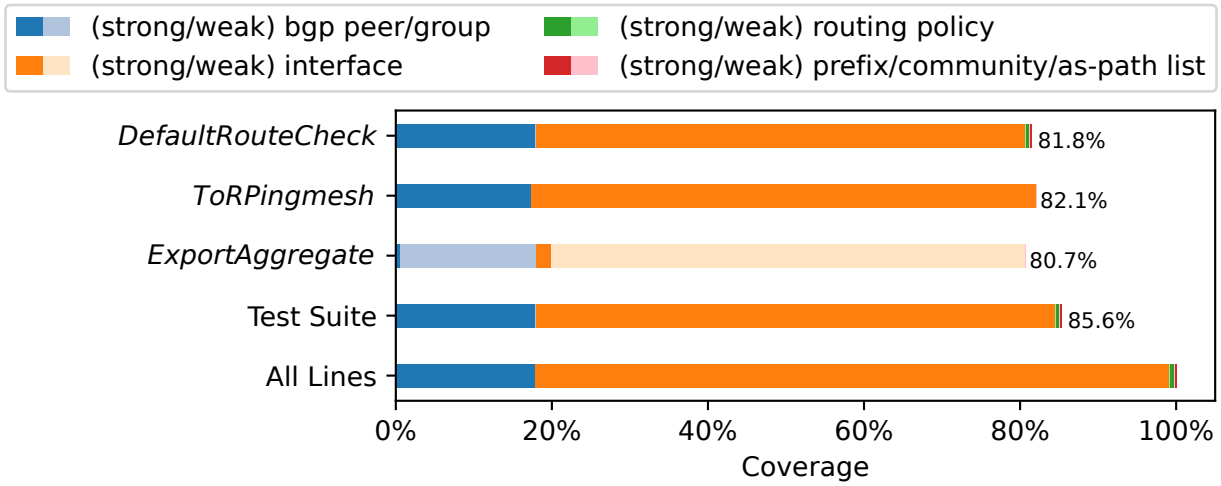


Figure 4.7: Coverage of synthetic datacenter network for different tests and types of configuration elements.

validation [42, 36].

- *DefaultRouteCheck*: ensure that each router has the default route.
- *ToRPingmesh*: ensure that each leaf router's assigned subnet is reachable from all other leaf routers.
- *ExportAggregate*: ensure that each spine router exports the aggregate route to WAN.

Figure 4.7 shows the coverage result when the network has a total of 80 routers. Given the uniformity of the network and the test suite, coverage results are similar for other network sizes. The total coverage of individual tests is 81.5%, 82.1% and 80.7% respectively, and the three tests together cover 85.3% of configuration lines. We find that these tests cover largely the same configuration elements—interfaces and BGP peerings between the data center routers—despite checking for seemingly different network behaviors. This result indicates that test development without coverage feedback can be ineffective in terms of covering the testing gaps.

The coverage of *ExportAggregate* shows a large proportion of weak coverage. This is because a spine router has routes to all leaf routers, so that all leaf subnets contribute to the tested aggregate route, albeit weakly. Separating out weak coverage here avoids false negatives of testing gaps—the aggregate routes would be there even if some of the BGP peering or interfaces are misconfigured, therefore testing the aggregate routes provides a weaker endorsement for the covered BGP peerings and interfaces to be bug-free.

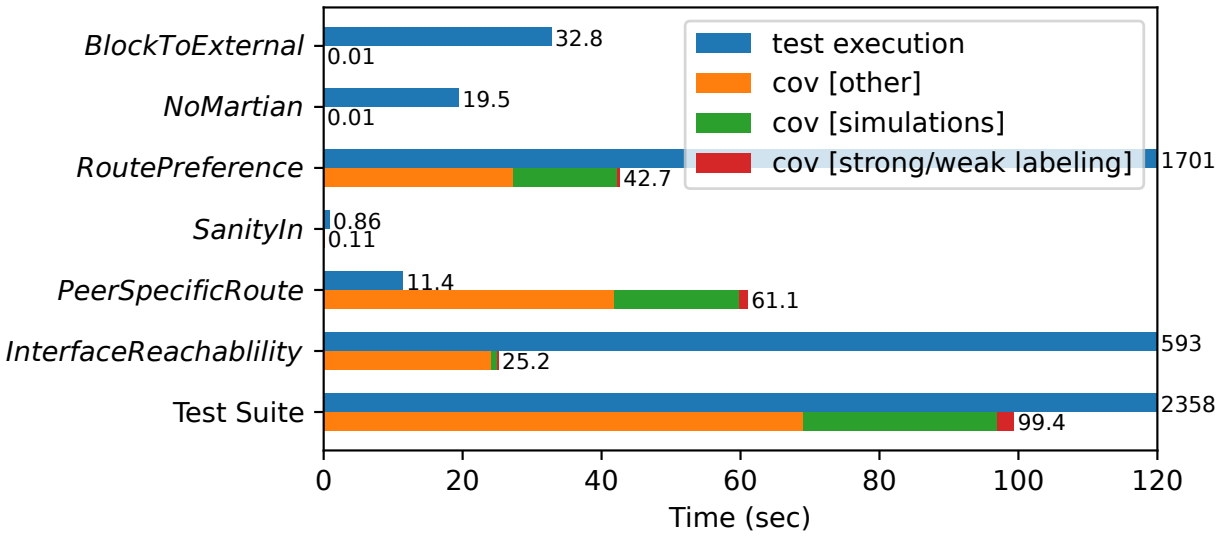
By looking at uncovered configuration lines reported by NetCov, we learn that most correspond to host-facing interfaces on leaf routers. Adding tests that target those interfaces improves this test suite and eliminate testing gaps. We omit results of this iteration.

4.5 Performance Evaluation

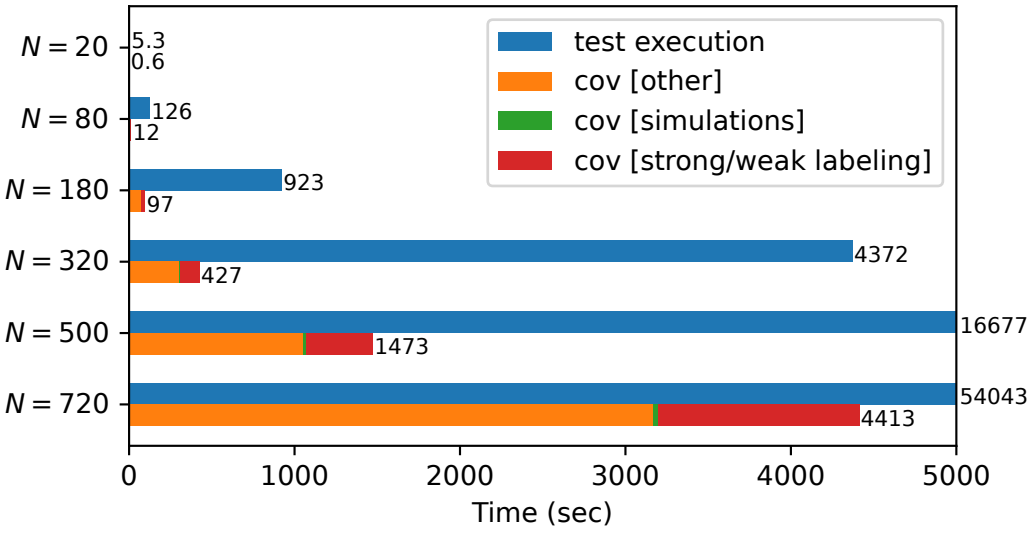
We benchmark the performance of NetCov on both types of networks we studied above. Our test machine has two Intel Xeon CPUs (16 core each, 3.1 Ghz), 384 GiB of DRAM, and runs Ubuntu 18.04.

Figure 4.8a shows the time to compute coverage for each test in §4.4.1 and for the full test suite. It breaks out the time spent on simulations and strong/weak labeling, and, for reference, also shows the test execution time. We see that coverage computation is reasonably fast. The full test suite takes only 99.4 seconds. In comparison, the test execution takes 2,358 seconds. The total coverage computation time is less than the sum for individual tests because facts tested by multiple tests are tracked only once. The graph also shows that simulations and strong/weak labeling are a minority component, which means that most of the time is spent on walking the IFG and doing lookups in stable state for backward inference.

Figure 4.8b shows test execution and coverage computation time for the test suite in §4.4.2, as a function of the data center network size. Coverage computation takes 4,413 sec on the largest network, which has 2,040,624 RIB entries. This time is less than 9% of the time to execute the test suite. While substantial, we deem it acceptable in practice. Configuration coverage analysis can be run in the background, as code coverage is often



(a) Internet2.



(b) Fat-tree networks.

Figure 4.8: Time to compute coverage.

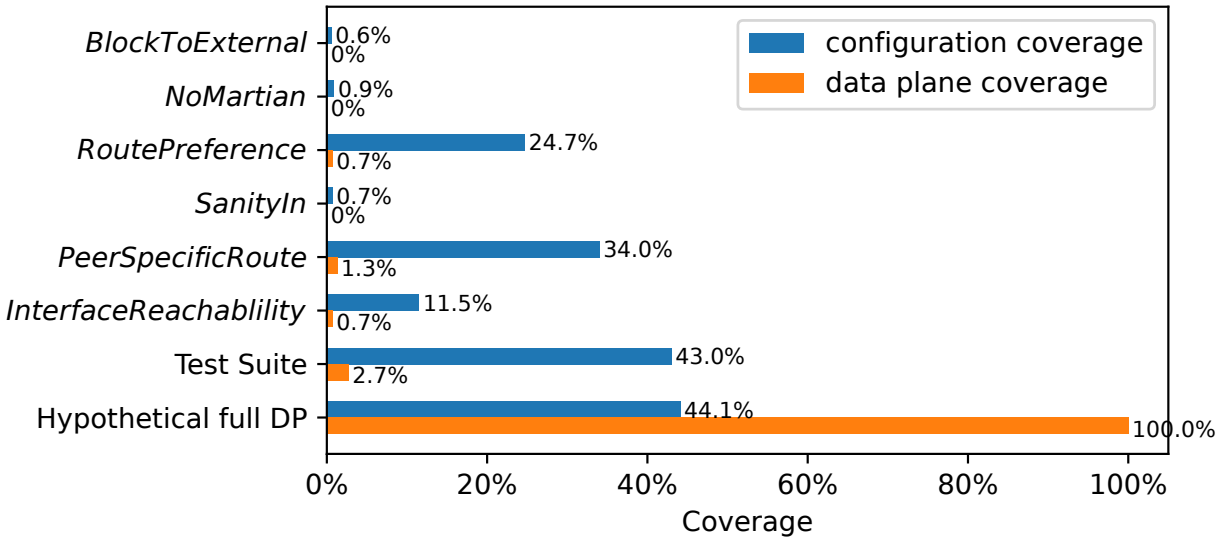
run. NetCov does not slow down test execution, which is on the critical path to finding configuration errors and updating the network.

However, time to compute coverage increases rapidly with network size. This is because the number of RIB entries grows quadratically and so does the number of vertices in the IFG. We find that the average time to materialize an IFG node does not change substantially because all computation is local to the node. The scaling trends suggest that to scale NetCov to much larger networks, we need a concurrent implementation of IFG materialization. Our current implementation is single-threaded (as Python interpreter is single-threaded).

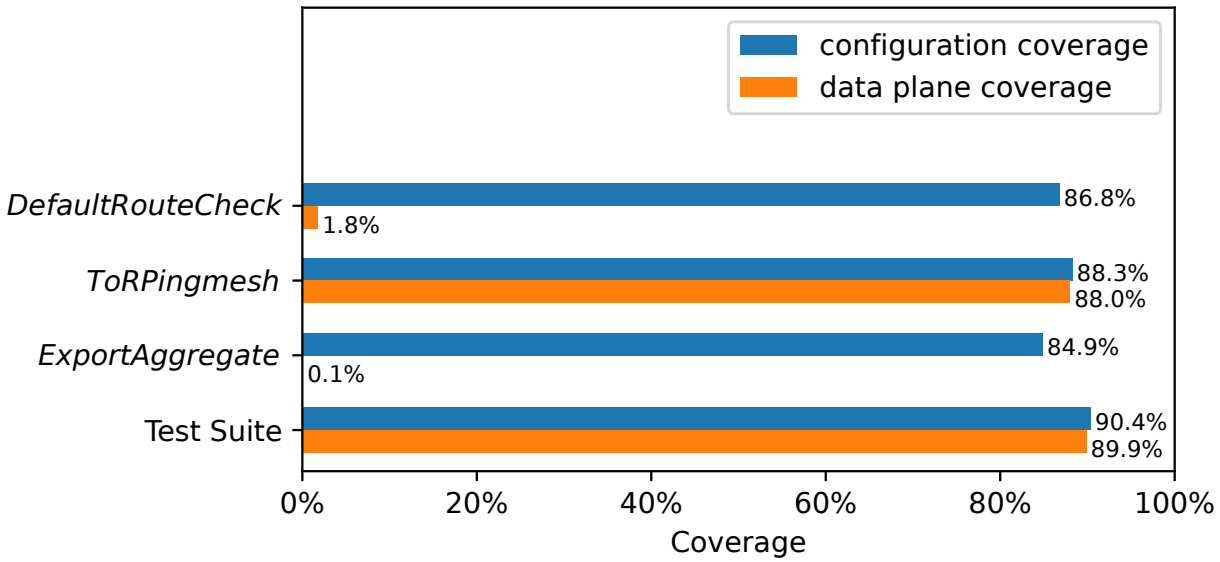
4.6 Comparison to Data Plane Coverage

We demonstrate the unique value of control plane coverage by comparing it to data plane coverage. Following Yardstick [78], we quantify data plane coverage as the proportion of main RIB (forwarding) rules exercised. Figure 4.9 shows the comparison for different cases. Figure 4.9a shows the comparison for Internet2 for all tests in §4.4.1 and a hypothetical data plane test that inspects all main RIB rules. Figure 4.9b shows the comparison for fat-tree tests in §4.4.2.

Besides the obvious advantage that only control plane coverage can support control plane tests—the graphs show 0% data plane coverage for these tests—there are two main advantages to using control plane coverage to guide network test development. First, it reveals testing gaps that can not be revealed by data plane coverage. Tests with high data plane coverage do not necessarily have high control plane coverage, as we can see in the last row of Figure 4.9a. Covering 100% of the data plane state covered only 41% of the configuration. If the engineers were to improve the test quality under the guidance of only data plane coverage, they would not know that 59% of the configurations remain untested. The reason of this disagreement is that some configuration lines are only exercised under specific environments (failures, routing messages). For instance, list-filtered route policies apply on BGP messages within a specific range, and will only be exercised when such



(a) Internet2.



(b) Fat-tree with $k = 10$.

Figure 4.9: Comparing control plane and data plane coverage.

messages appear in the environment.

Second, testing more data plane state can sometimes be redundant in covering configurations, when the tests hit the same configuration elements. For example, the *DefaultRouteCheck* test in Figure 4.9b has only 1.8% data plane coverage because it only tests default routes, which is a small fraction of all main RIB routes. However, because correct propagation of default routes incorporates many BGP peerings and interfaces in the network, this test has extensive configuration coverage (87%). The *ToRPingmesh* test covers much more data plane state (88%), but adding it atop *DefaultRouteCheck* has little value because this state is derived from almost the same set of configurations lines. We do not necessarily imply that engineers should drop one of these tests, as there may be other reasons to keep both. Our observations are about their value toward configuration coverage.

4.7 Related Work

Our work builds on top of four lines of research.

Code coverage. We borrow from the software domain the idea of using code coverage to reveal testing gaps, quantify test suite quality, and help engineers improve their test suites [39, 7, 32]. Our coverage analysis techniques, however, are specialized to the operation of network configurations.

Data plane coverage. Yardstick introduced data plane coverage metrics [78] that quantify the proportion of data plane elements such as forwarding rules and paths that are exercised by network tests. Configuration coverage goes further and maps tested data plane components to configuration elements that contribute to them. It provides more direct feedback because network engineers author configurations, not data plane state, and it supports testing of configuration elements that are not exercised by the current data plane state.

Network testing and verification. A range of tools can analyse properties of network data and control planes [36, 42, 45, 28, 80, 46, 38, 79, 13, 30]. NetCov borrows ideas from

verification tools to concisely model the network, *e.g.*, focusing on stable state and routing protocol instances [13, 30]. However, NetCov target a different problem—reveal what is tested vs enabling testing of new properties–and uses different techniques.

Network provenance. Provenance systems can track causal dependencies of events in distributed systems. Provenance systems like ExSPAN [84] materialize provenance graphs by tracing system execution in forward direction. Negative provenance systems can reason about missing events [77] and materialize provenance graphs lazily using backward inference. NetCov too uses a graph-based model. However, it is unique in terms of accommodating network configuration into a provenance model, and this model, tailored to the stable state assumption, is more succinct. Further, it combines backward and forward inference to overcome the limitations of using only one type of inference.

Software configuration testing. As for networks, configuration testing is an important problem for software systems as well. Sun et al. developed a system that can link software tests to exercised configuration parameters [64]. They exploit dependence on configuration settings being explicit, observable via read/write operations that use standard get/set APIs. NetCov targets a setting where the dependencies are implicit and non-local. Routers read the entire configuration file, and their forwarding behavior depends on that file and information received from neighbors who in turn act based on their configuration files and their neighbors. That led us to develop a different approach to tracking configuration dependencies. We will investigate in the future if our approach can be extended to software systems where dependence between tested runtime behavior and configuration is not explicit.

4.8 Conclusion

NetCov reveals which configuration lines are tested by a suite of network tests. It uses an information flow model based on control plane semantics to track which configuration lines contribute to tested data plane state. It accounts for non-local and non-deterministic

contributions, and for performance, it discovers the graph lazily. Our experiments showed that NetCov successfully reveals coverage gaps for real-world networks and test suites, and these tests can have surprisingly low coverage, e.g., 26% of configuration lines for Internet2. They also showed how its feedback helps improve coverage.

Stepping back, we note that networking is not alone in its reliance on configuration. Today, a lot of infrastructure and distributed applications are deployed by composing existing components using configuration (e.g., infrastructure deployment using Terraform, and application deployment using containers and service meshes). These configurations are central to correct behavior, which is why there is an intense focus on testing them properly [74, 40, 64]. As for networks, there are no tools to help engineers discover how well the configurations are tested. The techniques developed in our work, the IFG-based contribution tracking and its lazy traversal, can provide a starting point toward better testing of infrastructure and distributed application configuration as well.

Chapter 5

RELATIONAL NETWORK SPECIFICATION

Changing a running network, for instance, to alter its security posture, optimize resource usage, or add capacity, is one of the riskiest network management activities today. Outages can occur during changes because of incorrect change implementation (e.g., accidentally blocking traffic) or latent bugs (e.g., traffic starts traversing a longstanding filter). Erroneous changes have been a primary cause to numerous network outages and led to severe consequences [6, 67, 56, 57, 76, 59, 66]. Since changing a network is unavoidable, we must make changes safer to make networks more reliable.

The last decade has seen remarkable progress toward verification technologies that can reason about large, real-world networks. These technologies typically tell a user whether a *single* network snapshot N satisfies specification S. The snapshot may represent updated network configuration that engineers wish to deploy, and the specification may demand that DNS traffic is never blocked or that external traffic always traverses a firewall before reaching the high-security zone. Indeed, many large networks use these technologies today [20, 42, 8, 82].

Single-snapshot verification tools, while valuable, do not suffice for keeping networks running reliably as they are updated. Consider a common network change that moves all traffic on link A to link B as a precursor to shutting A for maintenance. To validate this change, the engineer would want to ensure that all traffic on link A is moved, that it is moved to link B and nowhere else, and that no other traffic is impacted. Using single-snapshot verification for this validation requires that the engineer (1) discover all traffic classes on link A, (2) create a specification asserting that the discovered traffic classes traverse link B in the new network, (3) discover *all other traffic classes* and *all their current paths*,

exactly, (4) create a specification asserting all such other traffic classes continue to follow these discovered paths.

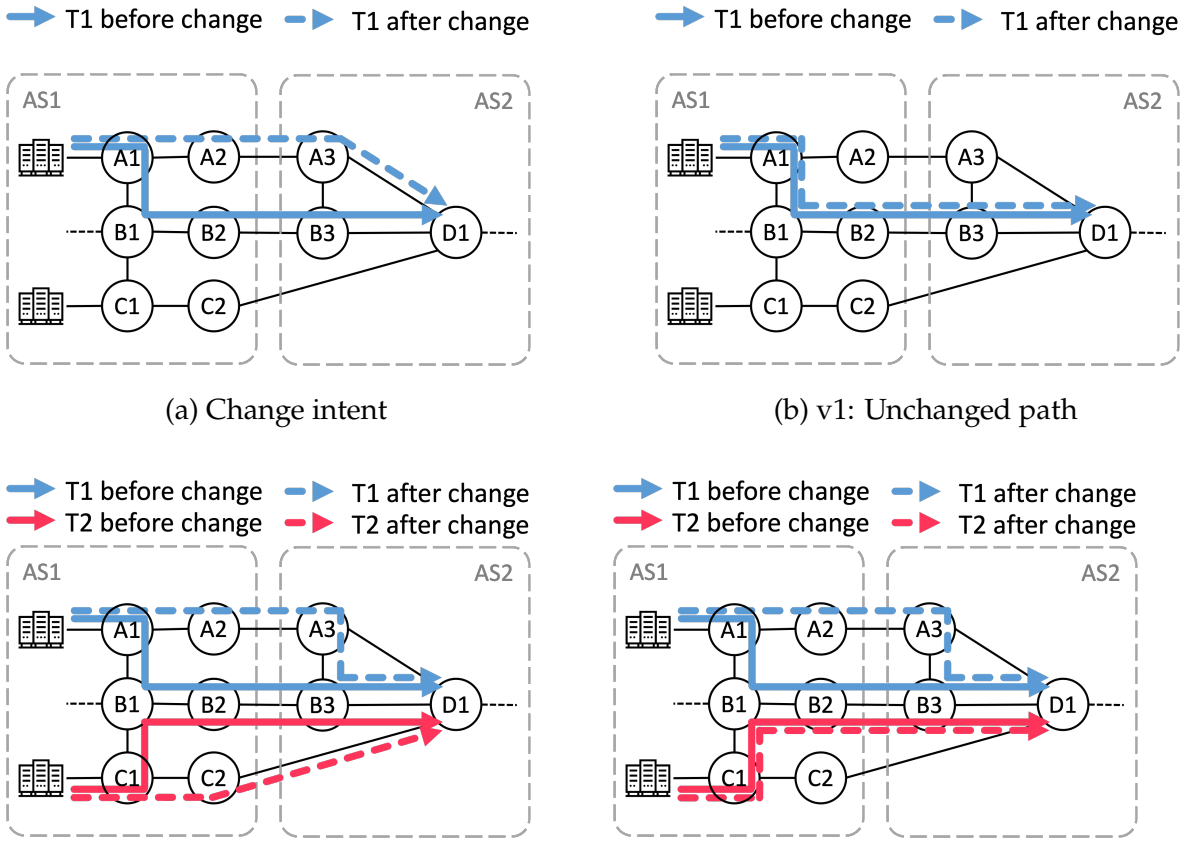
Unless the network is configured using high-level intents (e.g., Robotron [65]), which is rare today [20], creating such specifications is almost impossible. One challenge is *scale*: The specification needed scales with the size of the network, and modern networks are enormous and continue to grow. The network in our experiments has on the order of 10^3 routers, 10^4 routes per router, and 10^6 classes of flows with distinct forwarding paths, with up to 10^4 classes impacted by typical changes. Then there is an additional challenge of *incomplete information*: Networks evolve organically over years, and their size and complexity means an engineer may have only partial knowledge of a network's behavior. Creating precise, detailed specifications in these circumstances and maintaining them through successive changes requires otherworldly effort.

The upshot is that while single-snapshot verification helps ensure coarse, long-term invariants, it is not helpful when it comes to the fine details of many network updates. Yet network engineers must check such details to prevent congestion-induced outages, security breaches, or performance issues. Lacking appropriate tools, network engineers today rely on manually inspecting the impact of changes. Unsurprisingly, manual inspection is time-consuming, tedious, and error-prone, sometimes taking weeks to check even simple-seeming changes. See §5.1 for an example.

We introduce *relational network verification* and investigate if it can make network changes more reliable, more efficient, and less dependent on manual audits. Rather than reasoning about the behavior of a *single* snapshot in isolation, relational network verification reasons about the similarities and differences (i.e., the *relationships*) in the behavior of *two* network snapshots.

5.1 Network Changes Today

Implementing network changes requires that engineers translate their network-wide intents into specific device-level configuration changes. Unfortunately, errors in translation



(a) Change intent

(b) v1: Unchanged path

(c) v2: Collateral damage

(d) v3: Still incorrect path

Figure 5.1: An example network change in a global WAN. T1 and T2 denote aggregate traffic bundles.

between high-level intent and low-level implementation are common. Using a change from Alibaba Cloud's backbone, we illustrate the difficulty of making even seemingly simple changes and how incomplete information and scale limit the effectiveness of existing network analysis tools.

5.1.1 An Example Change

Figure 5.1a shows a change in the global backbone of Alibaba Cloud: The blue (solid) line denotes a path in the old network, and the orange (dotted) line denotes the new path desired for that traffic. Despite the simplicity of this abstract picture, it took network engineers *four iterations across three weeks* to devise a working implementation of the change.

The part of the backbone shown here has two BGP autonomous systems, $AS1$ and $AS2$, each enclosed by a grey box and had many routers. Each circle denotes a group of routers that fulfill the same functionality. An AS spans multiple geographic regions, encoded using the prefix letter of router groups. So, $A1$ and $A2$ are in the same region, which is different from that of $B1$ and $B2$.

The goal of the change is to prevent traffic, denoted $T1$, from region A from traversing region B while on its way to region D. In order to do so, all traffic on the path $A1$-$B1$-$B2$-$B3$-$D1$ should move to $A1$-$A2$-$A3$-$D1$. Importantly, though the picture does not specify this intent overtly, no other WAN traffic should be impacted.

First iteration. The engineers' first iteration (Figure 5.1b) changed the configuration of $A2$ routers. They added $T1$ prefixes to an allow-list on $A2$, with the hope that $A1$ would pick the shorter path $A1$-$A2$ over $A1$-$B1$-$B2$. However, on inspecting the impact of the change (using the process in §5.1.3), the engineers found it ineffective: The $T1$ traffic followed the same path as before! Investigation revealed that the routers in region B were configured to announce T1 prefixes with a high local preference. Since local preference overrides path length in BGP, $A1$ continued to prefer the route through $B1$ over $A2$. This failure illustrates the challenge of *incomplete information*: The engineers who implemented this

configuration change on $A2$ were not necessarily familiar with configuration details for $B2$ routers, as each configuration is accumulated from years of small changes made by different individuals.

Second iteration. The engineers' second iteration (Figure 5.1c) reconfigured $A2$ to increase the local preference of $T1$ prefixes. As a fail-safe, they also configured routers in region B to lower the local preference for these prefixes. This time, the engineers observed that $T1$ had indeed moved from $B2$ to $A2$. However, it turned out that the implementation caused *collateral damage*: the path of traffic T2, which should not have been impacted, changed. Debugging revealed that the root cause was a typo in the import policy at $B2$.

Third iteration. The next iteration (Figure 5.1d) fixed the typo. Upon testing, the engineers saw that it fixed the collateral damage but found another issue. While T1 traffic had indeed moved away from $B2$, it was bouncing back to $B3$ (due to an old configuration bug that made the link costs of $A3 - B3 - D1$ lower than those of $A3 - D1$). It turns out that this undesirable behavior was present in the last implementation as well, but the engineers missed it amidst the information overload created by the collateral damage. This failure illustrates the challenge of *scale*: It is not enough to focus on one small set of paths because even small configuration changes can impact many paths at once.

Fourth iteration. The fourth iteration finally achieved the intended behavior, after three weeks of labor.

Changes like this one are common in the backbone's daily operation. While some errors are caught prior to deployment, others make it to the network and have widespread impact.

5.1.2 Just Verify It?

Readers familiar with network verification might ask: Do the backbone network engineers have access to a verification tool; and does it help find errors in changes? The answers are: Yes, they have a verification tool, and they use it for certain tasks; and no, it does not help

uncover the types of errors above. We explain why.

Abstractly, existing network verification methods operate as follows: Given a specification S, check whether a network configuration C satisfies S. We call this method *single-snapshot verification* because it analyzes a single network configuration against the specification. Typically, one analyzes the new (post-change) network configuration, and the original (pre-change) network configuration is not used.

Single-snapshot verifiers are deployed to check coarse properties that hold over long periods of time, such as "never block DNS traffic" and "always block ssh from outside." These verifiers can validate that such properties are not violated by a proposed change, even when networks are very large. However, to catch finer-grained problems, such as the problems with specific paths, described in the previous subsection, much finer-grained specifications are needed. Unfortunately, with 10^6 traffic classes, creating a detailed specification for all of them is an insurmountable barrier to even getting started with single-snapshot network verification. Said differently, the cost of creating network-wide single-snapshot specifications is proportional to the size of the network. To make verification worthwhile, we need to create specifications at a lower cost, ideally proportional to the size of the change.

One naive tactic to address this challenge is to generate small but highly incomplete single-snapshot specifications. For instance, if a network engineer wishes to replace a path $P1$ with $P2$, they might verify that $P2$ exists in the new network and $P1$ does not. But this tactic omits a key property: all other traffic should remain unchanged, and hence does not help identify any collateral damage that may have occurred.

For these reasons, while single-snapshot verification has a role in ensuring network reliability, it is insufficient for change validation. Our backbone's network engineers thus resort to other methods, which we discuss next.

5.1.3 *Back to Manual Inspection*

The dominant change validation method that engineers use today is manual inspection. Its workflow is:

1. use a simulator [80, 28] to compute the network's forwarding state $N1$, based on the current configuration;

2. compute the network's after-change forwarding state $N2$, based on planned changes to the configuration;

3. use $N1$ and $N2$ to compute the before- and after-change forwarding paths for all flows that traversed the network over the last hour;[1] a *flow* is a packet that starts at a particular point in the network;

4. aggregate flows into *flow equivalence classes* that contain flows with identical paths in each before- and after-change configuration;

5. manually inspect the *path diff*, which contains all equivalence classes whose paths differ for the two configurations, and check that all expected changes have occurred and no unexpected changes have occurred.

Manual auditing is a tedious, mind-numbing affair subject to human error. Of course, the difficulty of conducting an audit depends in large part on the size of the path diff. Unfortunately, the path diffs can vary anywhere from tens of differences to over 10,000. Experienced engineers can audit only tens of classes per day, which makes a complete audit intractable for some changes. Engineers may thus have to resort to sampling, increasing the risk of missing problems. Further, while it is relatively easy (but still hard) to ensure that no undesired path changes occur by inspecting the path diff, ensuring that

[1] NetFlow [69] monitoring provides this data. Engineers prefer it over considering all possible flows (i.e., symbolic analysis) because it reduces information they need to inspect and helps focus on flows that matter.

all desired path changes occur is harder. Spotting an omission from a path diff is more difficult than missing the presence of a bad change.

5.2 A New Approach: Relational Verification

Relational network verification is inspired by today's manual approach and relational program verification research from the formal methods community (see Barthe [9] for an introduction). Relational methods reason about similarities and differences in *two* versions of a system, rather than considering one version in isolation. Because changes to a network involve two network configurations, one old and one new, these methods naturally apply.

Relational verification is better suited to validating network changes than single-snapshot verification because it is relatively easy to construct precise yet compact specifications for changes, even in enormous networks. Abstractly, to replace a path P_1 with P_2, a relational specification will declare that traffic flowing over P_1 in the old network should flow over P_2 in the new network and that all other traffic should follow the same path(s) in both networks. Such a specification takes just a few lines of code because relationally specifying "no change" (i.e., old equals new) is trivial. Indeed, the specification for the example change in the previous section is roughly 10 lines of code even though it moved over 10^4 flows. Importantly, "no change" specifications are inherently relational—they make direct use of comparisons between old and new—and there is no single-snapshot analog.

5.3 Rela by Example

Rela has a new specification language to describe the relationship between the forwarding behavior of two network snapshots. This section introduces the language using the change in Figure 5.1. The next section formalizes its syntax and semantics.

Recall that the intent of the change in Figure 5.1 has three elements: (1) only impact the traffic from region A to D that traverses $A1$ and $D1$; (2) change the forwarding sub-paths of this traffic from $A1$-$B1$-$B2$-$B3$-$D1$ to $A1$-$A2$-$A3$-$D1$, while leaving unchanged the sub-paths

before $A1$ and after $D1$ (which may be unknown to the engineer making the change); (3) no other traffic should be impacted.

Change Zones. In Rela, the first step in defining a change intent is to define the *change zone*. Informally, change zones allow users to create a focus area for the impact of a change and ignore behaviors outside of that focus. Users define change zones using *path patterns*, which are regular expressions over network locations.

A *network location* identifies one hop in a forwarding path. In Rela, forwarding paths and locations can be viewed at different levels of granularity, including at the interface level, the router level or the router group level. Users may choose the level of granularity that suits their needs. Our example uses router-level locations; our user does not care which interfaces are used for forwarding as long as they belong to the correct router.

Rela is used in concert with a database that stores information about all locations available in the network. Users can refer to a set of locations within the same entity (such as a router group or a tier) by issuing "where" queries to select locations from the database and return the union of them. We define below `a1` to be the set of routers with group attribute $A1$. A similar query defines `d1`.

```
regex a1 := where(group=="A1")
regex d1 := where(group=="D1")
```

Regular expressions `a1` and `d1` can now be used to refer to routers in $A1$ and $D1$ in the rest of the Rela specification. For instance, the regex `a1.*d1` denotes the set of paths that starts from any location in $A1$ and ends at any location in $D1$ after traversing zero or more (any) intermediate locations.

Change specifications. An atomic *change specification* is written *zone* : *modifier*. Roughly speaking, such a specification indicates that paths in the zone should be changed according to the modifier. When desired, such specifications may be named and reused or composed with other change specifications. For instance:

```
spec name := { zone : modifer; }
```

Path modifiers describe the sets of paths to add, remove, replace, or preserve between old and new network snapshots. For example, the following code presents one implementation of the second element of our example change intent.

```
spec pathRepl := {
    a1.*d1 : replace(a1b1b2b3d1, a1a2a3d1);
}
```

This spec has zone `a1.*d1`, which cares about the subset of pre-change forwarding paths that start from `a1` and end in `d1`. Its modifier demands that if one or more paths in the zone is matched by regex `a1b1b2b3d1`, then all paths in regular path set `a1a2a3d1` should appear in the new network (assuming symbols `a2`, `b1`, etc., have all been defined earlier as the union of routers in the corresponding router group). The semantics of **`replace`** also demands that (1) if any path in regular path set `a1a2a3d1` appears in the old network, it continues to appear in the new network, and (2) paths that belongs to zone `a1.*d1` but not belongs to regex `a1b1b2b3d1` should remain the same throughout the change.

We note that the **`replace`** modifier demands *all* paths in `a1a2a3d1` appear in the new network snapshot if any path in `a1b1b2b3d1` appears. This may be what the user wants in some cases, but it may not be in others. After all, `a1a2a3d1` represents the Cartesian product of four router groups and contains a large number of possible paths—does the user want all such paths to be present in the new network? The initial informal English specification we gave is actually mute on this issue; it simply says "change it." Indeed, we have found that working with Rela requires we think very carefully about exactly what we require, and typically, there are many corner cases to consider. Still, because the specifications are so short (as well as reuseable and re-executable), one can afford to think carefully about their consequences.

Fortunately, Rela provides several different built-in modifiers if **`replace`** is not the desired one. If the traffic should move to *some* path ("any" of them) in `a1a2a3d1`, an engineer can use the **`any`**`(regex1)` modifier, as follows.

```
spec pathShift := {
    a1.*d1 : any(a1a2a3d1);
}
```

Recall that traffic in our change zone may start upstream of $A1$ routers and continue downstream of $D1$ routers. The spec above has not expressed changes expected for these starting and ending *sub-paths*. The user may not even know all the paths leading to this part of the network. In other systems, specifying a change accurately with such incomplete information is challenging, or perhaps impossible. Fortunately, though, Rela is *compositional* as well as relational: One may stitch together change specifications of different kinds for different subpaths to construct an end-to-end specification. In this case, to specify that the beginnings and ends of our paths should not change, we can use change specifications with the `preserve` modifier as follows.

```
spec e2e := {
    a* : preserve;
    pathShift;
    d* : preserve;
}
```

This spec, which concatenates three atomic specs, defines the change zone as "`a* (a1.*d1 ) d*`". The first sub-spec's zone is `a*`, which denotes arbitrary length paths within region A. Even though users may not know the details of sub-paths in this zone, they do understand that these sub-paths are expected to remain unchanged, and the `preserve` modifier does the trick. We then reuse `pathShift` defined earlier to specify the sub-path changes in the middle. And the spec of the third and last sub-path is similar to the first one. Rela thus allows a precise end-to-end spec to be expressed compositionally, even when some parts of the paths are unknown to the users.

Up to this point, we have a spec that defines which paths should change and how they should change. Our third and final task is to specify that no other paths are affected by

the network update. Once again, Rela makes this task easy via composition of specs using the >> operator:

```
spec nochange := { .* : preserve; }
spec change := e2e >> nochange
```

All traffic that does not match the first spec will fall through to the next spec chained by >>. Thus, all existing traffic except those matched by `e2e` will be required to comply with `nochange`—it must stay the same.

Summary. Rela specifications describe relations between a pair of network snapshots—that is, the paths that are added, removed, replaced or preserved when an old network is updated. It allows change zones to be defined at a level of location granularity appropriate to their task. Once a zone of interest is defined, one may craft atomic change specifications that describe the relation between old and new networks for (sub-)paths in a zone. Users may draw on a collection of pre-defined modifiers to define relations of interest. Finally, complex change specs may be built out of simple ones through the use of Rela's composition operators.

5.4 Formalizing Rela specifications

This section specifies the formal syntax of Rela and provides its semantics via translation to an intermediate representation with *regular relations*, which we call the RIR. While the RIR is more expressive than Rela's surface language, it is low-level, making it harder to use by network engineers. Indeed, Rela was created with a goal of making it easier to write relational specifications for networking use cases. Still, an expert user may use the RIR directly if they choose.

5.4.1 Rela Syntax

Figure 5.2 presents Rela's formal syntax, which includes sub-languages for (regular) sets of paths (r), modifiers (m), simple specifications (s), header constraints (h) and guarded

Path Sets	r	::=	a
		|	$(r_1 \mid r_2)$
		|	$r_1 r_2$
		|	$r*$
Modifiers	m	::=	**preserve**
		|	**add**(r)
		|	**remove**(r)
		|	**replace**(r_1, r_2)
		|	**drop**
		|	**any**(r)
Simple Specs	s	::=	$r : m$
		|	$s_1 s_2$
		|	s_1 `>>` s_2
Header Constraints	h	::=	$true$
		|	`dst in` $prefix$
		|	`src in` $prefix$
		|	`dscp ==` $digits$
		|	$(h_1 \mid h_2)$
		|	h_1 `&` h_2
		|	`!`h
Guarded Specs	g	::=	s
		|	`if(`h`) {`g`}`
		|	`if(`h`) {`g_1`}` **else** `{`g_2`}`

Figure 5.2: The syntax of Rela's front-end language.

specifications (g). The paths are built from a set of locations Σ, which includes a single special location $drop$—that place (akin to /dev/null) where intentionally dropped packets go. We use a to range over elements of the set Σ. This syntax omits named definitions **spec** $name$:= { g } and **regex** $name$:= { r }, which are easily inlined. It also excludes **where** queries to select locations from database, which are implemented as a prepass.

We saw several of the modifiers (m) in the previous section. One that we did not see is **drop**, which replaces old paths with a new path that drops a packet. Each modifier is defined by a straightforward translation into the RIR. While our experiments suggest that we have developed a useful set of modifiers, new ones can be added by encoding their semantics in the RIR.

A simple specification (s) defines an expected change between a set of old paths and a set of new paths in a network. Simple specs include path modifiers ($r : m$), concatenation of simple specs ($s_1 s_2$), and prioritized union of simple specs ($s_1 >> s_2$).

Sets of packets are described by header constraints (h). In our current implementation, programmers may use the source IP (e.g., **src in** *prefix*), destination IP, or DSCP (a field in the IP header that encodes the priority of the packet) to describe packet sets.

Guarded specifications (g) describes changes for the paths taken by different subsets of packets. For example, the conditional statement `if(`h`) {`s`}` specifies that when packets described by h take paths M in the old network and N in the new network, M and N should be related by s. This specification says nothing about the paths taken by packets outside the set described by h. More concretely, consider the common change of decommissioning an IP prefix. For this change, we want to remove the paths used by traffic to the target prefix, but preserve the paths for all other traffic, even if the other traffic traverses similar or identical sets of paths. We can encode this decommissioning requirement for address $10.0.0.0/24$ using the following specification.

```
spec deallocP :=
  if (dst in 10.0.0.0/24) {
```

$$
\begin{array}{lcl}
P \in \mathit{Path\ Set} & ::= & a \mid 0 \mid 1 \mid \texttt{PreState} \mid \texttt{PostState} \mid (P_1|P_2) \mid P_1P_2 \mid P^* \\
& \mid & P_1 \cap P_2 \mid P_1 \setminus P_2 \mid \overline{P} \mid P \triangleright R \\
R \in \mathit{Relation} & ::= & P_1 \times P_2 \mid \texttt{I}(P) \mid 0 \mid 1 \mid (R_1|R_2) \mid R_1R_2 \mid R^* \mid R_1 \circ R_2 \\
S \in \mathit{Simple\ Spec} & ::= & P_1 = P_2 \mid P_1 \subseteq P_2 \\
G \in \mathit{Guarded\ Spec} & ::= & S \mid h \mapsto G \mid G_1 \wedge G_2
\end{array}
$$

Figure 5.3: RIR syntax

```
  .* : remove(.*);
} else {
  .* : preserve;
}
```

5.4.2 *Regular IR (RIR)*

The Rela RIR is an intermediate language for defining *regular sets* of paths and *regular relations* between paths. A regular set is a set created through the usual operations on regular languages (concatenation, union, and Kleene star). Likewise, regular relations are binary relations between paths (i.e., sets of pairs of paths), also constructed with the usual operations on regular languages. Since all RIR-expressible sets and relations are regular, we are able to make use of known, efficient constructions and decision procedures from automata theory as the basis of a decision procedure for RIR.

Figure 5.3 presents the syntax of the RIR, which contains four sub-languages. The language of path sets (P) describes regular sets of paths over the alphabet Σ (as defined in §5.4.1). The path sets a, 0, and 1 denote sets with a single one-hop path, no paths at all, and a single 0-length path (written ϵ). The special symbol `PreState` denotes the set of paths in the pre-change network. Similarly, `PostState` denotes the set of paths in the post-change network. The expressions $P_1|P_2$, P_1P_2, and P^* denote union, concatenation,

(a) Path Sets

$$\mathscr{P}[\![a]\!](M,N) \triangleq \{a\}$$

$$\mathscr{P}[\![0]\!](M,N) \triangleq \emptyset$$

$$\mathscr{P}[\![1]\!](M,N) \triangleq \{\epsilon\}$$

$$\mathscr{P}[\![\texttt{PreState}]\!](M,N) \triangleq M$$

$$\mathscr{P}[\![\texttt{PostState}]\!](M,N) \triangleq N$$

$$\mathscr{P}[\![P_1 \mid P_2]\!](M,N) \triangleq \mathscr{P}[\![P_1]\!](M,N) \cup \mathscr{P}[\![P_2]\!](M,N)$$

$$\mathscr{P}[\![P_1 P_2]\!](M,N) \triangleq \{p_1 p_2 \mid p_1 \in \mathscr{P}[\![P_1]\!](M,N), p_2 \in \mathscr{P}[\![P_2]\!](M,N)\}$$

$$\mathscr{P}[\![P^*]\!](M,N) \triangleq \{p_1 ... p_n \mid p_1, ..., p_n \in \mathscr{P}[\![P]\!](M,N)\}$$

$$\mathscr{P}[\![P_1 \cap P_2]\!](M,N) \triangleq \mathscr{P}[\![P_1]\!](M,N) \cap \mathscr{P}[\![P_2]\!](M,N)$$

$$\mathscr{P}[\![P_1 \setminus P_2]\!](M,N) \triangleq \mathscr{P}[\![P_1]\!](M,N) \setminus \mathscr{P}[\![P_2]\!](M,N)$$

$$\mathscr{P}[\![\overline{P}]\!](M,N) \triangleq \Sigma^* \setminus \mathscr{P}[\![P]\!](M,N)$$

$$\mathscr{P}[\![P \triangleright R]\!](M,N) \triangleq \{q : \exists p.\ \langle p, q\rangle \in \mathscr{R}[\![R]\!](M,N) \wedge p \in \mathscr{P}[\![P]\!](M,N)\}$$

(b) Relations

$$\mathscr{R}[\![P_1 \times P_2]\!](M,N) \triangleq \{\langle p_1, p_2\rangle \mid p_1 \in \mathscr{P}[\![P_1]\!](M,N), p_2 \in \mathscr{P}[\![P_2]\!](M,N)\}$$

$$\mathscr{R}[\![\mathtt{I}(P)]\!](M,N) \triangleq \{\langle p, p\rangle \mid p \in \mathscr{P}[\![P]\!](M,N)\}$$

$$\mathscr{R}[\![0]\!](M,N) \triangleq \emptyset$$

$$\mathscr{R}[\![1]\!](M,N) \triangleq \{(\epsilon, \epsilon)\}$$

$$\mathscr{R}[\![R_1 | R_2]\!](M,N) \triangleq \mathscr{R}[\![R_1]\!](M,N) \cup \mathscr{R}[\![R_2]\!](M,N)$$

$$\mathscr{R}[\![\mathtt{I}(P)]\!](M,N) \triangleq \{\langle p, p\rangle \mid p \in \mathscr{P}[\![P]\!](M,N)\}$$

$$\mathscr{R}[\![R_1 R_2]\!](M,N) \triangleq \{\langle p_1 p_2, q_1 q_2\rangle \mid \langle p_1, q_1\rangle \in \mathscr{R}[\![R_1]\!](M,N), \langle p_2, q_2\rangle \in \mathscr{R}[\![R_2]\!](M,N)\}$$

$$\mathscr{R}[\![R^*]\!](M,N) \triangleq \{\langle p_1 ... p_n, q_1 ... q_n\rangle \mid \langle p_1, q_1\rangle, ..., \langle p_n, q_n\rangle \in \mathscr{R}[\![R]\!](M,N)\}$$

$$\mathscr{R}[\![R_1 \circ R_2]\!](M,N) \triangleq \{\langle x, z\rangle \mid \exists y. \langle x, y\rangle \in \mathscr{R}[\![R_1]\!](M,N), \langle y, z\rangle \in \mathscr{R}[\![R_2]\!](M,N)\}$$

(c) Simple Specs

$$M, N \models P_1 = P_2 \iff \mathscr{P}[\![P_1]\!](M, N) = \mathscr{P}[\![P_2]\!](M, N)$$

$$M, N \models P_1 \subseteq P_2 \iff \mathscr{P}[\![P_1]\!](M, N) \subseteq \mathscr{P}[\![P_2]\!](M, N)$$

(d) Guarded Specs

$$F, M, N \models S \text{ if } F = \emptyset, \text{ and otherwise:}$$

$$F, M, N \models S \iff M, N \models S$$

$$F, M, N \models h \mapsto G \iff F \cap \mathscr{H}[\![h]\!], M, N \models G$$

$$F, M, N \models G_1 \wedge G_2 \iff F, M, N \models G_1 \text{ and } F, M, N \models G_2$$

Figure 5.4: RIR semantics.

and Kleene star operations over path sets. Finally, $P \triangleright R$ denotes the *image*, the path set derived by applying relation R to paths recognized by P. In other words, $P \triangleright R$ describes the set of paths that are related (via R) to *some* path recognized by P.

Figure 5.4 (a) presents the evaluation functions that define the semantics of path sets. These equations have the form $\mathscr{P}[\![P]\!](M, N) \triangleq S$, meaning that P describes the set of paths S when M is the pre-change set of forwarding paths and N is the post-change set of forwarding paths.

Relations in Figure 5.3 denote regular relations, which are sets of pairs of paths. Alternately, a relation may be viewed as a map from each path in the domain to zero or more related paths in the image. The cross-product relation $P_1 \times P_2$ denotes the relation that associates every path in P_1 with all paths in P_2. The identity relation $\mathtt{I}(P)$ associates every path in P with itself; paths not in P are not related to any other path by $\mathtt{I}(P)$. The symbols 0 and 1 denote the empty relation and the relation associating ϵ with itself. $R_1|R_2$, R_1R_2, R^*, and $R_1 \circ R_2$ denote union, concatenation, Kleene star, and composition of relations. (Rational relations are closed under all of these operations [26].)

Figure 5.4 (b) shows the semantics of relations. The equations have the form $\mathscr{R}[\![R]\!](M, N) \triangleq T$, meaning that R describes set of pairs of paths T when M is the pre-change set of forwarding paths and and N is the post-change set of forwarding paths.

Simple specs (Figure 5.3) may be equations ($P_1 = P_2$) or subset relations ($P_1 \subseteq P_2$). As an example, consider this spec:

$$\texttt{PreState} \triangleright R = \texttt{PostState}$$

Assuming the relation R is an intended transformation of network forwarding paths, the spec says that if one applies the transformation R to the pre-change path set, then one should obtain a result that equals the post-change path set. Our translation from Rela's surface language into the RIR uses this sort of idiom.

Figure 5.4 (c) presents the semantics of simple specs. The satisfaction relation has the form $M, N \models S$, which may be read as "pre-change forwarding paths M and post-change

forwarding paths N satisfy S"

Finally, guarded specs (Figure 5.3) allow programmers to specify that the paths travelled by different sets of packets should change in different ways. Guarded specs (G) include simple specs (S) as well as restricted specs ($h \mapsto G$) and conjunctions thereof.

Figure 5.4 (d) shows the semantics of guarded specs in terms of *flow equivalence classes* (FECs), which are triples of the form (F, M, N) where F is a set of packets, M is a set of paths that F follows in the pre-change network, and N is a set of paths that F follows in the post-change network. The semantics depends upon a function $\mathscr{H}$, which maps header constraints to the set of headers that satisfy it. For example, the constraint `dst in 10.0.0.0/24` specifies the set of packets with destination IP prefix in the given range. We omit the full definition of $\mathscr{H}$ for brevity. In general, the satisfaction relation for guarded specs has the form $F, M, N \models G$, which may be read as "flow equivalence class (F, M, N) satisfies the guarded specification G." An FEC (F, M, N) satisfies a spec G whenever F is an empty set of packets, and if F is non-empty, satisfaction is defined by induction on the structure of G.

5.4.3 Compilation from Rela to RIR

In this subsection, we show how to compile any Rela simple spec expression (s) into an RIR simple spec (S) and any Rela guarded spec (g) into an RIR guarded spec (G).

Ultimately, a simple specification s is translated into an RIR expression of the following form.

$$\texttt{PreState} \triangleright \mathscr{R}_{pre}[\![s]\!] = \texttt{PostState} \triangleright \mathscr{R}_{post}[\![s]\!]$$

In this process, we generate two relation expressions from s. The first relation ($\mathscr{R}_{pre}[\![s]\!]$) transforms the pre-change paths and the second relation ($\mathscr{R}_{post}[\![s]\!]$) transforms the post-change paths.

In what follows, we show how to compute relations for some of the key modifiers in the Rela language.

Encoding path preservation. Consider the translation of the path preservation modifier "`D: preserve`". Intuitively, this change specification says that all paths that appear in the zone D in the pre-state should also appear in the post-state. If the pre- and post-relations are as follows:

$$\mathscr{R}_{pre}[\![D : \texttt{preserve}]\!] \triangleq \texttt{I}(D)$$
$$\mathscr{R}_{post}[\![D : \texttt{preserve}]\!] \triangleq \texttt{I}(D)$$

then our overall translation will be:

$$\texttt{PreState} \triangleright \texttt{I}(D) = \texttt{PostState} \triangleright \texttt{I}(D)$$

which is equivalent to the equation:

$$(\texttt{PreState} \cap D) = (\texttt{PostState} \cap D)\ ,$$

as desired.

Encoding path additions. Consider adding the paths P when the pre-change network contains a path in D.[2] Our goal now is to preserve all of the paths in the zone (and also P, if it exists) from the pre-state into the post-state. In other words, we would like to apply the identity relation $\texttt{I}(D \mid P)$. In addition, we would like a relation that adds the path P. We can use the relation $D \times P$ to do so. Overall, our pre-relation is the combination of those two relations. Hence we generate the following equations.

$$\mathscr{R}_{pre}[\![D : \texttt{add}(P)]\!] \triangleq \texttt{I}(D \mid P) \mid (D \times P)$$
$$\mathscr{R}_{post}[\![D : \texttt{add}(P)]\!] \triangleq \texttt{I}(D \mid P)$$

Encoding path removals. Next, consider path removals using the modifier "`D: remove(P)`". This modifier expresses that the paths in D in the pre-state should be preserved in

[2]The Rela surface language can not express addition of a path in D when the pre-change network contains no path in D. Such "unconditional" path additions can be expressed in the RIR, however. For instance, the equation $\texttt{PostState} = \texttt{PreState}|P$ expresses that exactly the set of paths recognized by P are added to the network.

the post-state, except the paths in P which should be removed. Hence, our relations are as follows.

$$\mathscr{R}_{pre}[\![D : \texttt{remove}(P)]\!] \triangleq \texttt{I}(D \setminus P)$$
$$\mathscr{R}_{post}[\![D : \texttt{remove}(P)]\!] \triangleq \texttt{I}(D)$$

Encoding non-deterministic path replacement. The modifier "`D: any(P)`" demands that (1) if there is any path in $D \mid P$ in the pre-state, there must be some path in P in the post-state and (2) all paths in $D \mid P$ in the post-state must be in P. To encode this condition, we use a relation for the pre-state that replaces paths in $D \mid P$ with a symbol $\#$. Likewise, the relation for the post-state replaces all paths in P with $\#$, while also retaining the paths in $D \setminus P$. Since paths in $D \setminus P$ are *not* retained in the pre-state relation, this relation encodes that there are no paths in $D \setminus P$ in the post-state network. Together, the two relations enforce the desired condition.

$$\mathscr{R}_{pre}[\![D : \texttt{any}(P)]\!] \triangleq (D \mid P) \times \#$$
$$\mathscr{R}_{post}[\![D : \texttt{any}(P)]\!] \triangleq (P \times \#) \mid \texttt{I}(D \setminus P)$$

Encoding prioritized union. A prioritized union "$s_1 >> s_2$" should apply the change specification s_1 to s_1's zone and s_2 to everything else in s_2's zone. To achieve this specification in the RIR, we need to explicitly extract s_1's zone. We do so with an auxiliary function $\mathcal{Z}[\![D : \texttt{modifier}]\!]$. See Figure 5.5 for the full definition of $\mathcal{Z}[\![\cdot]\!]$.

To translate "$s_1 >> s_2$", we first translate s_1, and then take the union with the translation of s_2 applied exclusively to the complement of the zone of s_1.

Summary of simple specs. See Figure 5.5 for the complete translation for simple specs.

Translation for guarded specs. A guarded spec is simply an if-then-else structure that applies different header constraints to different simple specs. Based on the translation of

$$\mathcal{S}[\![s]\!] \triangleq \texttt{PreState} \triangleright \mathcal{R}_{pre}[\![s]\!] = \texttt{PostState} \triangleright \mathcal{R}_{post}[\![s]\!]$$

$$\begin{aligned}
\mathcal{R}_{pre}[\![D : \texttt{preserve}]\!] &\triangleq \texttt{I}(D) \\
\mathcal{R}_{pre}[\![D : \texttt{add}(P)]\!] &\triangleq \texttt{I}(D \mid P) \mid (D \times P) \\
\mathcal{R}_{pre}[\![D : \texttt{remove}(P)]\!] &\triangleq \texttt{I}(D \setminus P) \\
\mathcal{R}_{pre}[\![D : \texttt{replace}(P_1, P_2)]\!] &\triangleq \texttt{I}((D \mid P_2) \setminus P_1) \\
&\quad \mid ((D \cap P_1) \times P_2) \\
\mathcal{R}_{pre}[\![D : \texttt{drop}]\!] &\triangleq (D \mid drop) \times drop \\
\mathcal{R}_{pre}[\![D : \texttt{any}(P)]\!] &\triangleq (D \mid P) \times \# \\
\mathcal{R}_{pre}[\![s_1 s_2]\!] &\triangleq \mathcal{R}_{pre}[\![s_1]\!]\ \mathcal{R}_{pre}[\![s_2]\!] \\
\mathcal{R}_{pre}[\![s_1 >> s_2]\!] &\triangleq \mathcal{R}_{pre}[\![s_1]\!] \mid \left(\texttt{I}(\overline{\mathcal{Z}[\![s_1]\!]}) \circ \mathcal{R}_{pre}[\![s_2]\!]\right)
\end{aligned}$$

$$\begin{aligned}
\mathcal{R}_{post}[\![D : \texttt{preserve}]\!] &\triangleq \texttt{I}(D) \\
\mathcal{R}_{post}[\![D : \texttt{add}(P)]\!] &\triangleq \texttt{I}(D \mid P) \\
\mathcal{R}_{post}[\![D : \texttt{remove}(P)]\!] &\triangleq \texttt{I}(D) \\
\mathcal{R}_{post}[\![D : \texttt{replace}(P_1, P_2)]\!] &\triangleq \texttt{I}(D \mid P_2) \\
\mathcal{R}_{post}[\![D : \texttt{drop}]\!] &\triangleq \texttt{I}(D \mid drop) \\
\mathcal{R}_{post}[\![D : \texttt{any}(P)]\!] &\triangleq (P \times \#) \mid \texttt{I}(D \setminus P) \\
\mathcal{R}_{post}[\![s_1 s_2]\!] &\triangleq \mathcal{R}_{post}[\![s_1]\!]\ \mathcal{R}_{post}[\![s_2]\!] \\
\mathcal{R}_{post}[\![s_1 >> s_2]\!] &\triangleq \mathcal{R}_{post}[\![s_1]\!] \mid \left(\texttt{I}(\overline{\mathcal{Z}[\![s_1]\!]}) \circ \mathcal{R}_{post}[\![s_2]\!]\right)
\end{aligned}$$

$$\begin{aligned}
\mathcal{Z}[\![D : \texttt{preserve}]\!] &\triangleq D \\
\mathcal{Z}[\![D : \texttt{add}(P)]\!] &\triangleq D \mid P \\
\mathcal{Z}[\![D : \texttt{remove}(P)]\!] &\triangleq D \\
\mathcal{Z}[\![D : \texttt{replace}(P_1, P_2)]\!] &\triangleq D \mid P_2 \\
\mathcal{Z}[\![D : \texttt{drop}]\!] &\triangleq D \mid drop \\
\mathcal{Z}[\![D : \texttt{any}(P)]\!] &\triangleq D \mid P \\
\mathcal{Z}[\![s_1 s_2]\!] &\triangleq \mathcal{Z}[\![s_1]\!]\ \mathcal{Z}[\![s_2]\!] \\
\mathcal{Z}[\![s_1 >> s_2]\!] &\triangleq \mathcal{Z}[\![s_1]\!] \mid \mathcal{Z}[\![s_2]\!]
\end{aligned}$$

Figure 5.5: Rela to RIR translation for simple specs ($\mathcal{S}[\![\cdot]\!]$).

simple specs, we define the following evaluation functions for guarded specs ($\mathcal{G}[\![\cdot]\!]$)

$$\begin{aligned}
\mathcal{G}[\![s]\!] &\triangleq \mathcal{S}[\![s]\!] \\
\mathcal{G}[\![\texttt{if}(h)\ \{g\}]\!] &\triangleq h \mapsto \mathcal{G}[\![g]\!] \\
\mathcal{G}[\![\texttt{if}(h)\ \{g_1\}\ \texttt{else}\ \{g_2\}]\!] &\triangleq h \mapsto \mathcal{G}[\![g_1]\!] \\
&\quad \wedge\ !h \mapsto \mathcal{G}[\![g_2]\!]
\end{aligned}$$

5.5 Decision procedure

Rela's decision procedure models a network change as a set of flow equivalence classes (FEC). Given an RIR spec and a network change, the decision procedure determines whether each FEC in the change meets the spec. If not all FECs meet the spec, an exhaustive list of counterexamples will be provided in the form of specific packets and paths that violate the spec.

5.5.1 Checking Guarded Specs

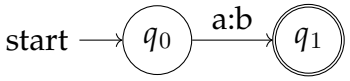
To validate a guarded spec against an FEC, we may follow the rules described in §5.4.2. Thus we are left with the problem of checking whether a pair of path sets M, N satisfies a simple spec.

5.5.2 RIR to Finite-State Automaton (FSA)

To validate an RIR simple spec regarding two sets of paths, we first construct a finite-state automaton (FSA) from each $Path\ Set$ and Rel expression in the simple spec. Per Kleene's Theorem, every regular language can be represented by an FSA that moves from one state to another in response to the input sequence of symbols. Similarly, every regular relation can be represented as a finite-state transducer (FST) [26].

An FST is essentially an FSA that uses two tapes. It may be viewed as a "translating machine" that reads from an input tape and writes to the output tape. The following

diagram presents an FST for relation $a \times b$, which translates path a into path b.



The label a:b on the arc means a should be read from the input tape and b should be written to the output tape.

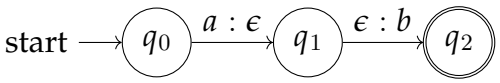
From small, simple FSAs, like the one above, we can build larger, more complex ones using standard automaton composition algorithms. In what follows, we sketch some of the algorithms used to construct Rela-specific symbols and operators. Most other aspects of the compilation strategy are well-established, and are thus omitted (see Thompson's construction [70], for instance).

`PreState` and `PostState` symbols. Conceptually, the input to the decision procedure contains two sets of forwarding paths that corresponds to `PreState` and `PostState` respectively. In practice, however, the number of ECMP forwarding paths may explode in a large network. This problem is prominent when forwarding behavior is modeled at the interface-level, because the network may employ 10s of parallel links between any two hops to increase capacity. Indeed, we recorded a flow with 10^8 interface-level ECMP paths for our backbone, and it takes several hours just to deserialize the paths from file input. To address this challenge, Rela defines a graph format to represent the interface-level input path set. Each vertex in the graph denotes a router that appears as a forwarding hop for this traffic, and each directed edge denotes a physical link that is used to forward this traffic between the two hops. There is also extra metadata to identify all source vertices and sink vertices (start and end locations of paths) in the DAG. With this format, the 10^8 paths of the aforementioned traffic class can be encoded with a DAG with 38 vertices and 50K edges.

Constructing an FSA for `PreState` and `PostState` from the forwarding graph is straightforward: We turn vertices and edges in the DAG into FSA states and transitions respectively. If the user has specified a coarser granularity than interface-level (e.g., router level),

we do granularity conversion in this step by merging vertices that belong to a same coarser entity. Next, we add a initial state q_0 and draw an ϵ-transition from q_0 to each source node identified by the metadata. Finally, we set all sink nodes to be accepting states of the FSA.

$P_1 \times P_2$ **relation.** The FST for $P_1 \times P_2$ may be obtained by (1) translating the FSA for P_1 into an FST that accepts P_1 on its first tape and ϵ on its second, (2) translating the FSA for P_2 to a FST that accepts P_2 on its second tape and ϵ on its first, and (3) concatenating the results. An illustration of this construction for $a \times b$ can be found in figure below (recall that ϵ is the empty string).



$\mathtt{I}(P)$ **relation.** The FST of $\mathtt{I}(P)$ is the same as the FSA of P except that each FST transition has an output symbol that duplicates the input symbol on the same transition.

$P \triangleright R$ **image.** To obtain the FSA of $P \triangleright R$, one may compute the composition of relations $\mathtt{I}(P)$ and R, and then compute the output projection of $\mathtt{I}(P) \circ R$. The output projection of an FST removes the input symbols from all transition arcs (while keeping the output symbols) to derive an FSA from an FST. The composition of two regular relations $R_1 \circ R_2$ may be compiled using standard FST algorithms [26].

5.5.3 Compliance Checking

Once we have the FSA representation of both sides of an equation ($P_1 = P_2$ or $P_1 \subseteq P_2$ in simple specs), we can check for equality (or inclusion) using standard automaton equivalence algorithms.

5.5.4 Counterexample Generation

If previous steps found any FEC that violates an RIR specification, then we generate an exhaustive list of counterexamples, where each entry contains a flow equivalence class (FEC)

Flows	Pre-change paths	Post-change paths	Cause of violation
$\{(pkt = (\dots), start = x_1)\}$	$\{x_1A_1B_1B_2B_3D_1y_1\}$	$\{x_1A_1A_2A_3B_3D_1y_1\}$	`e2e`: $\{x_1A_1A_2A_3D_1y_1\} \neq \{x_1A_1A_2A_3B_3D_1y_1\}$
$\{(pkt = (\dots), start = x_2)\}$	$\{x_2C_1B_1B_2B_3D_1y_2\}$	$\{x_2C_1C_2D_1y_2\}$	`nochange`: $\{x_2C_1B_1B_2B_3D_1y_2\} \neq \{x_2C_1D_2D_1y_2\}$

Table 5.1: A subset of counterexamples generated by Rela when verifying the change implementation in Figure 5.1c. The first row shows a flow in traffic class T1, and the second row shows a flow in T2.

and a reason that explains the failure. An FEC contains a set of flows (a tuple of a packet and a starting location) with the same before- and after-change paths. Table 5.1 shows a subset of counterexamples reported by Rela when verifying the change implementation in Figure 5.1c using the `change` spec in §5.3. The two entries indicate incorrect path changes for traffic T1 and collateral damage for T2.

The forwarding paths that violate a simple spec are derived by extracting paths from the difference of two FSAs. Recall that a Rela simple spec s is translated to an equation of the form $P_1 = P_2$ in RIR, where $P_1 = \texttt{PreState} \triangleright \mathcal{R}_{pre}[\![s]\!]$ and $P_2 = \texttt{PostState} \triangleright \mathcal{R}_{post}[\![s]\!]$. The difference $P_1 \setminus P_2$ represents the expected forwarding paths that are missing from the observed post-change network, and $P_2 \setminus P_1$ represents the unexpected paths in the post-change network.

For each violating FEC, we generate a reason to help understand why it failed the spec. For simple specs that are composed using the >> operator, we can find the exact sub-spec that failed a flow by matching the flow with the zone of each sub-spec. We then apply $\mathcal{R}_{pre}$ and $\mathcal{R}_{post}$ of this sub-spec to the flow's pre- and post-change path set respectively. The difference of the two derived sets explains the failure of set equation and inclusion assertions made by the spec. For special symbols introduced by rewriting in the compilation process, we rewrite them back to their original forms to make the counterexamples more human-readable. For example, the before paths in the first row of Table 5.1 yield

$\{x_1 \# y_1\}$ when applying $\mathcal{R}_{pre}$, where "#" rewrites $A_1A_2A_3D_1$. After undoing this rewriting, the "Cause of violation" column clearly shows that the flow failed the sub-spec `e2e`, which expected the path set to be $\{x_1A_1A_2A_3D_1y_1\}$ after change. This set is not equal to the observed path set $\{x_1A_1A_2A_3B_3D_1y_1\}$.

5.6 Implementation

We implemented Rela with 6,000 lines of Python code. Rela and RIR are implemented as domain-specific languages embedded in Python. The decision procedure uses the OpenFST library [3] and the Python bindings provided by HFST [48] to construct and compose finite state automata and transducers. We implemented certain automaton operations, such as the product relation ($P_1 \times P_2$), ourselves using low-level HFST APIs that manipulate automata directly.

For each packet equivalence class, Rela reads the before and after forwarding paths from file input, which is produced by the same network simulation toolchain described in §5.1.3. We tweaked the simulator to output forwarding paths in the Rela-defined graph format compactly and to enable efficient FSA construction for `PreState` and `PostState` expressions (§5.4.3). Each packet equivalent class is processed in parallel.

The Rela source code is publicly available [62].

5.7 Qualitative Evaluation

We evaluated Rela qualitatively and quantitatively. This section presents the qualitative evaluation, which focuses on user experience. The next section presents a quantitative evaluation of Rela's expressiveness and performance. For the qualitative evaluation, we ran Rela on historical changes to Alibaba Cloud's backbone. Our workflow shared the first four steps with the current workflow in §5.1.3: simulate pre- and post-change networks, compute forwarding paths, aggregate them into equivalence classes. The final step is different: the forwarding data is given to Rela as input, along with a spec, and we analyze all equivalence classes rather than just the diff. We describe how this process played out for

the change in §5.1.1, compared to the original experience of the engineers, and then draw lessons from our experience.

5.7.1 Revisiting the Example Change

For each proposed change (i.e., "iteration"), we used Rela to check the change against a relational specification.

First iteration. We invoked Rela with the change implementation v1 (Figure 5.1b) and the `change` spec in §5.3. For this implementation, the path diff of the manual inspection tool had 17 flow equivalence classes. Engineers investigated each class and discovered that none of them corresponded to the desired path change, and all of them stemmed from either issues with the simulation tool or benign side effects of the change. The allow-list change on $A2$ routers caused unexpected but acceptable traffic changes.

Rela produced 17 counterexamples for `nochange` and 15 for `e2e`. The 15 violations for `e2e` clearly signaled that the change failed to move T1 traffic, as the pre-change and post-change paths were still the same for such flows. The counterexamples for `nochange` are the same as those reported by the path diff tool. To automatically exclude such benign violations in future iterations (and avoid triaging them again), we extended the spec with a new component called `sideEffects`, to explicitly permit such changes.

Second iteration. In the second iteration, we provided the change implementation v2 (Figure 5.1c) and the refined spec. For this implementation, the current path diff tool produced a path diff with 46 classes. Engineers waded through them to discover the collateral damage and, because of information overload, missed that the change to T1 traffic was incorrect.

Rela produced 15 counterexamples for `e2e`, 24 for `nochange`and 0 for `sideEffects`. The violations signaled that changes to T1 traffic was wrong and that there was collateral damage as well. The refined `sideEffects` spec helped suppress benign differences.

Final iteration. Because Rela discovered two errors at the same time, we skipped the third

iteration (which was needed during the original manual analysis), and jumped straight to the final iteration. In this iteration, we supply the correct change implementation to Rela and the refined spec. Rela validated the change automatically and completely. In contrast, in their original debugging effort, the engineers had to manually inspect the path diff to certify the change.

5.7.2 Lessons Learned

Based on our experience with Rela, we draw these lessons:

1. Rela's categorization of violations based on which sub-spec is violated speeds up error diagnosis and reduces the number of iterations. Errors are quickly diagnosed because the violated sub-spec provides strong hints about the root cause; the types of errors that violate `nochange` are different from those that violate `e2e`. The number of iterations is reduced because multiple errors in an implementation are easier to spot, especially when spread across different sub-specs. With manual inspection, when analyzing a big bag of path diffs, it is hard to spot multiple errors.

2. Rela specs may need refinement because the original change intent (in natural language) is under-specified or the network is not configured as expected. Under-specification and unexpected behaviors are common for large networks. However, while the effort put into a manual audit is hard to reuse, effort put into refining a Rela spec pays off. The refined spec saves work during future iterations of the same change or other similar changes. Multiple changes of the same type are a common occurrence for production networks.

3. When a change (sub) spec does not match an implementation, there is less data to analyze. Change implementations are often partially correct, and Rela produces only violations. The current path diff contains both compliant and non-compliant changes. The engineers must analyze both to find violations.

4. When the change spec matches the implementation, the engineers need to do nothing. They can have greater confidence in the change compared to manually inspecting the path diff.

5.8 Quantitative Evaluation

To quantify the expressiveness and performance of Rela, we apply it to a set of real network changes in the global backbone of Alibaba Cloud. This dataset has all changes that were reviewed by the network's technical committee from Jun 2023 to Jan 2024. The committee reviews all high-risk, complex changes. There are 10s of changes in the dataset. We omit the exact count for confidentiality.

5.8.1 Expressiveness

We used Rela to specify engineers' intent for each change in the dataset. We determined the intent by examining change tickets, which contain a description of the intent in natural language as well as a change implementation plan. The tickets describe change intents pertaining to the network data plane as well as those of other types, such configuration settings and backup routes. We focused on data plane change intents. All changes in our dataset have a data plane change intent, and three in four have only data plane change intents. We list the Rela specs of a subset of evaluated changes along with intent descriptions in Appendix A.1.

We found that Rela can specify the intended data plane change for 97% of the changes in our dataset. That Rela can support this high a fraction of high-risk changes in a large, complex network is a highly encouraging indicator of its expressiveness.

For the remaining 3% of the changes, Rela could only partially specify the intended data plane change. Rela's key current limitation is a lack of support for path counting: In addition to path shape, users sometimes want to limit the number of paths that a flow can take. For example, because of router hardward limits, one might not want the number

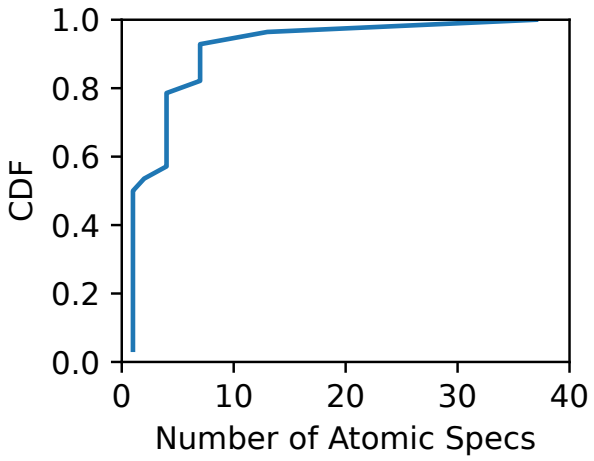


Figure 5.6: Distribution of spec size in our dataset.

of ECMP (equal cost multipath routing) paths for a flow to exceed 128. We will explore supporting such intents in the future by generalizing the `any` modifier to include a path count.

To assess the compactness of specifications, we quantify their size as the number of atomic Rela specs (of the form `r : m`). This analysis excludes any spec refinement that may be needed to accommodate benign side effects (§5.7.1); we do not have the data to make that determination. Figure 5.6 shows a cumulative distribution function (CDF) of the number of atomic specs needed across all changes. The vast majority of the changes (93%) can be expressed with fewer than 10 atomic specs. The outliers correspond to infrequent changes to the backbone's routing architecture in which significant traffic carried by the network is shifted.

Half the changes require only one atomic spec, corresponding to no expected impact on the forwarding behavior. It may seem odd at first that so many high-risk changes fall in this bucket. But fully preserving forwarding behavior while something is changed under the hood (e.g., modifying the routing policy to replace concrete routes with aggregate routes or standardizing on community tags) is common. It is also high-risk. Indeed, there

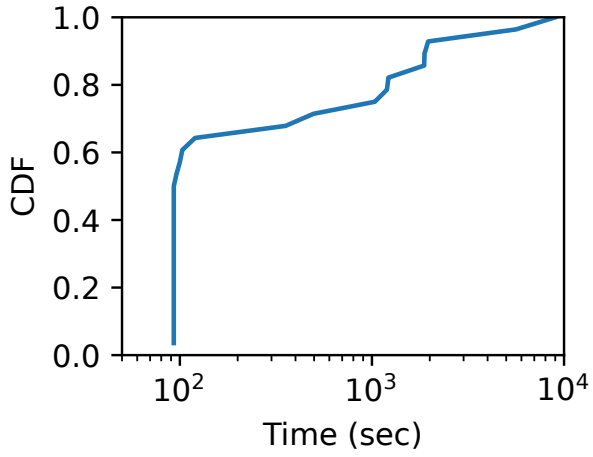


Figure 5.7: Time (log scale) to validate changes with Rela.

are changes in our data where no behavior change was expected but the path diff revealed forwarding changes that could have led to an outage.

5.8.2 Performance

We benchmark Rela's performance using the time to validate changes in our dataset, including the time to deserialize the forwarding path data, FSA/FST construction and equivalence checking. This experiment was done on a computer with 96 CPU cores and 768 GB DRAM. Because we did not have access to the precise data plane states of historical changes, we ran all specs on the same data plane state produced by a recent snapshot. We release a subset of this data plane state [62]. Guidance for accessing it is in Appendix A.2.

Figure 5.7 shows a CDF of the validation time. Half of the changes take 93 seconds, which is the time to check the "no change" spec. Four in five changes need less than 20 minutes, and the most complex change needs 150 minutes. For context, we observe that it takes 140 minutes to simulate both network snapshots and compute forwarding paths. We conclude from these results that the performance of Rela is acceptable for the backbone network, especially considering how long manual inspection takes today.

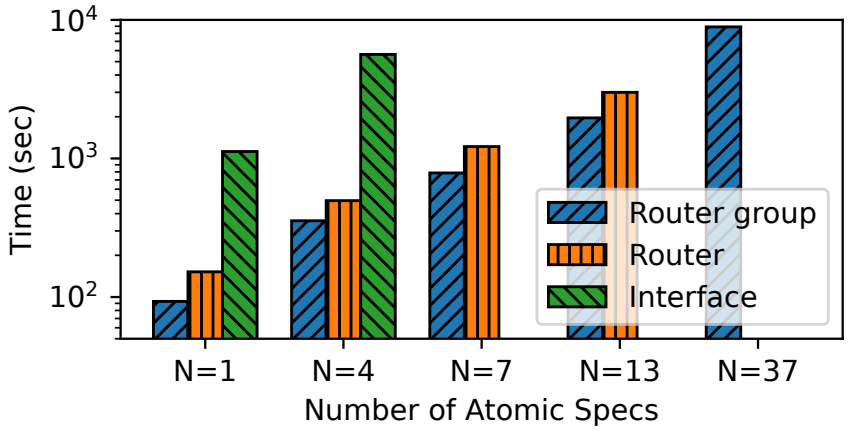


Figure 5.8: Rela's validation time (log scale) for different spec sizes and location granularity.

Diving deeper into Rela's performance, we find that the two most important factors are the size of the spec (number of atomic specs) and the location granularity. Figure 5.8 shows this impact by running specs in our data at different granularities. (Figure 5.7 used the granularity indicated by the change intent, so it has a mix.) We exclude granularity-size combinations that need over 3 hours.

We see that validation time grows with the spec size, and finer granularity analysis takes more time (as expected). The impact of going from the router group level to the router level is small, but the impact of going to the interface level is substantial (10x), due to the substantially higher number of paths at the interface level. Fortunately, under 4% of the changes in our data require interface-level granularity. 7% require device-level.

5.9 Related Work

Our work builds on the foundation laid by single-snapshot verification tools [52, 45, 28, 46, 79, 42, 80, 13, 30, 27, 1, 54, 5]. The application of these tools to real-world networks has improved reliability and provided insights into the problems they do and do not solve.

We act upon one such insight: that many large, real-world networks are difficult to specify accurately in their entirety. Without such single-snapshot specifications, engineers need different kinds of tools to help them validate network changes automatically.

Differential network analysis (DNA) [83] shares our perspective on network verification—that it is crucial to track similarities and differences between pre- and post-change networks. It simulates the pair of pre- and post-change control planes efficiently to generate differences in their data plane states. (Rela makes no contributions to control plane simulation.) In addition to showing path diffs, DNA can generate differences in single-snapshot invariants, e.g., "A can reach B in the pre-change network but not the post-change network." Engineers must manually inspect the path and invariant diffs to determine whether or not they indicate errors. In contrast, Rela specifications characterize what constitutes an error, and our decision procedures check these specifications automatically. Importantly, Rela's specifications can be perfectly precise, more precise than "A can reach B"—any specific path or regular set of paths may be specified. This precision takes manual audits completely out of the loop when changes are conformant.

Batfish supports differential analysis as well [11]. It independently analyzes two snapshots and formats the output such that the differences are easier to analyze. Like DNA, it requires humans to certify correctness and does not have a relational spec. Once again, Rela improves on this situation using a relational specification language and deciding the validity of specifications without human auditing.

Our language design is inspired in part by NetKAT [5], which has shown that using regular languages (Kleene algebra) is an effective way to specify network behavior. Rela builds on this idea and uses regular relations in addition to regular languages to express differences and similarities between pairs of networks.

Researchers have explored relational verification for ordinary programs many times in the past [10, 17, 21]. The archetypal goal here is to verify that two programs are equivalent. At least superficially, the techniques for relational program verification differ from those in Rela. A common method is to consider a "product" program that combines two input

programs and verify the safety properties of this product. An interesting avenue for future work is to consider whether specific relational program verification techniques can help us verify networks more efficiently or vice versa.

5.10 Summary

We develop the concept of relational network verification and realize it in the Rela tool for validating network changes. Our key observation is that relational specifications can compactly and precisely capture change intents; they need only express what is expected to change, which is often a miniscule fraction of the overall network, and simply say "no change" for the rest. For a global backbone with over 10^3 routers, 93% of the high-risk changes need fewer than 10 terms and 80% of them can be validated in under 20 minutes.

While we focus on change verification, we expect that relational reasoning for networks will prove effective in other contexts as well. For instance, it could help verify if two parts (e.g., two geographic regions) of the *same* snapshot are similar modulo a few exceptions. It could also compactly describe a large network for the purposes of synthesis by describing a baseline network behavior and local modifications to the behavior. We look forward to exploring other applications of relational network analysis.

Chapter 6

LIMITATIONS AND FUTURE WORK

6.1 Limitations

The work in this dissertation made progress towards the challenge of writing complete and precise network specifications. However, limitations remain when applying such techniques to diverse network designs and various scenarios of network operations.

6.1.1 Unsupported Network Features

The development of both Yardstick and NetCov started from considering a small subset of network features. Yardstick considers data planes that are stateless and only use destination-based forwarding. The extensions to support stateful network devices (such as firewalls) and other forwarding mechanisms (such as label-based and policy-based forwarding) are possible, but the interplay of these features will significantly increase the testable space and thus the computation cost. Their influences on the scalability of coverage computation are concerning and need further investigation.

NetCov's information-flow model and on-demand inference are suitable for connected routes and BGP, two most commonly used routing protocols. But we have not investigated whether it can be extended to link-state protocols such as OSPF. One of the open problems in such extensions would be to determine weather each network link contributes to a shortest path, because one path being selected implicitly depend on other links being set high costs. NetCov needs extensions to support label-switching protocols as well.

6.1.2 *From Concrete to Symbolic*

Concrete vs symbolic analysis are two complimentary techniques to check the correctness of network routing and forwarding. Concrete methods provide explainable and more relevant results (avoids false positives for packets that will never be carried by the network), while symbolic ones may have performance gains when a huge amount of possibilities need to be analyzed anyways.

To this regard, NetCov's approach supports control plane analysis only for concrete operating environments. It is an interesting open problem to extend the IFG model to symbolic forms.

Rela currently employs concrete data plane simulation in the toolchain to compute forwarding paths for concrete input packets given the data plane states. If we were to analyze Rela specifications for symbolic header spaces, one possible way is to employ a symbolic data plane simulator that can compute the forwarding paths of all flow equivalence classes in the network. This process would be prohibitively expensive in complex networks. There are opportunities to implement a more efficient decision procedure if we could prune the state-exploration process in symbolic data plane simulation using hints from Rela specs. This direction is yet to be investigated.

6.1.3 *Beyond Relational Path Properties*

We have shown in Rela that relational path properties can be used to concisely specify multiple kinds of real-world network changes. However, it is far from an one-fits-all solution given the variety of network operation scenarios. For example, some changes have explicit intents regarding the network's routing behaviors that cannot be reflected in forwarding paths (such as adding a backup route as a fail-safe). Some other changes may care about that the network links are not overloaded in additional to that the correct forwarding paths are being used. In such scenarios, engineers may wish to write specifications that are beyond path properties. Moreover, relational specifications cannot replace single-snapshot

specification in the entire life-cycle of network management. When a new network is built upfront, or when radical changes are made to a significant proportion of the network, engineers still better non-relational languages to express the desired network behaviors.

6.2 Future Work

6.2.1 Generative Network Specifications

While coverage metrics and relational languages can make it easier for engineers to write high-quality specifications, both techniques still involves heavy human labor in the loop. Given the coverage results, it is engineers' responsibility to review the under-specified aspects in the network and propose specification refinements; With the Rela language, it is still engineers who decide which exact specifications to use. One interesting future work is to explore automatic specification generation and refinement using Generative Artificial Intelligence (GenAI). Such tools can generate various kinds of new contents from input data and instructions, and have rapidly gained widespread usage [25]. In software industry, GenAI has shown promising improvement of productivity in various tasks such as test generation from code and code generation from instructions. However, a well-known challenge of using GenAI is to handle the possible false information in the output, which in our scenario, is to ensure the completeness and precision of generated specifications. We look forward to exploring the potential of using GenAI for network specification and enhancing generative network specifications with coverage analysis and appropriate specification languages.

6.2.2 Toward Intent-Driven Network Operations

High-quality network specifications make it possible to realise the intent-driven networking paradigm, where the design, validation, deployment and maintenance of networks are all largely autonomous processes driven by network intents. This dissertation has demonstrated the use of Rela specifications in before-deployment validation of network changes.

In the future, we look forward to exploring the use of relational specifications to streamline other phases of network operations, such as change implementation and runtime testing.

Change implementation today involves writing configuration commands in vendor-specific syntax or generating them from pre-defined templates to realise the desired change effects. For changes whose desired effects can be completely and precisely captured by relational specifications, it is now possible to synthesis correct-by-design configuration implementations. Unlike synthesizing a new network from scratch using single-snapshot specifications [15], relational synthesis needs to take into account the current state of network configurations and generate changes on top of it. This is a new problem yet to be formalized.

Runtime testing of network changes refers to the process of probing network behaviors before, during and after the deployment and check that the observed network behaviors match the expectations in each stage. Runtime testing complements pre-deployment verification as the latter may miss errors related with hardware bugs or network dynamics. Such aspects are usually not modeled by network simulators or emulators. One practical challenge of runtime testing is the lack of synchronous global view of network state and the limited amount of probing that can be made to avoid over-utilization of network resources. The formalization and solution for this problem is an interesting future work.

Chapter 7

CONCLUSION

Network specifications formulate the expected behavior of networks so that automated testing and verification tools can check correctness and prevent bugs before they cascade into costly network outages. However, it is challenging to write complete and precise specifications for today's large and complex networks using the abstractions provided by existing tools and without help on specifying completeness. The poor quality of specifications has significantly constrained the effectiveness of testing and verification efforts and ultimately undermines the reliability of the network infrastructure [16, 58, 63].

This dissertation presented two techniques to help network engineers write complete and precise specifications.

Network Coverage Metrics. The first technique is coverage analysis to help network engineers assess whether different aspects of network behaviors have or have not been covered by existing specifications. Such coverage feedback simplifies the identification of underspecified components and thus knowledge of additional specifications needed.

We developed two families of network coverage metrics, i.e., data plane and control plane. For network data plane coverage, we formalized the atomic testable unit of stateless network data planes, enabling us to unify the definition and computation for a range of coverage metrics to support various data plane components and diverse test types. For network configuration coverage, we formalized the dependency model between data plane table entries and the configuration lines that contribute to the production of each individual entry. We also developed an efficient algorithm to infer such dependencies on demand. Both data plane and configuration coverage have revealed systematic testing gaps in production networks that were unnoticed previously.

Relational Specification. The second technique is the relational abstraction of network change specifications. Unlike single-snapshot specifications that address the behavior of one network snapshot, relational specifications model the similarity and differences (*i.e.*, the relations) between two network snapshots (*i.e.*, before and after changes). The relational abstraction helps network engineers address the challenges of scale and incomplete information because they need to specify only the changing parts of network behavior and isolate them from others that do not change. The changing parts are usually small and known to the engineers who implement the changes.

We developed the first relational network specification language, Rela, that describes (1) the changing parts of network forwarding behavior as regular expressions over network locations and (2) the desired ways of making changes using a set of common modifiers. We further developed an intermediate representation layer, well-grounded in the established theories of rational relations, and a technique to lower Rela specifications to the IR. The IR design enables efficient verification using finite-state automata. Rela has demonstrated not only its compactness and expressiveness in historical changes from the backbone network of a large cloud provider, but also significant productivity improvements for the overall validation practice of network changes.

These results notwithstanding, we believe that many challenges in this area remain. The scale of networks are ever growing. New routing and forwarding mechanisms are being adopted. Full automation of network design, validation and deployment are still beyond the horizon. We expect the journey of developing new tools and new languages for high-quality network specification will continue.

BIBLIOGRAPHY

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *Proceedings of NSDI 20*, pages 201–219. USENIX Association, 2020.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of SIGCOMM '08*, page 63–74. ACM, 2008.

[3] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. Openfst: A general and efficient weighted finite-state transducer library: (extended abstract of an invited talk). In *Implementation and Application of Automata: 12th International Conference, CIAA 2007, Praque, Czech Republic, July 16-18, 2007, Revised Selected Papers 12*, pages 11–23. Springer, 2007.

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of POPL '14*, page 113–126. ACM, 2014.

[6] Mae Anderson. Time Warner cable says outages largely resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved, accessed on June 23, 2021, 2014.

[7] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[8] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, et al. Reachability analysis for AWS-based networks. In *International Conference on Computer Aided Verification*, pages 231–241. Springer, 2019.

[9] Gilles Barthe. An introduction to relational program verification. https://software.imdea.org/~gbarthe/__introrelver.pdf, accessed on Feb 2, 2024, 2020.

[10] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, 2011.

[11] Differential questions, 2022.

[12] Batfish: Network configuration analysis tool. https://github.com/batfish/batfish.

[13] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of SIGCOMM '17*, pages 155–168. ACM, 2017.

[14] Ryan Beckett and Ratul Mahajan. Putting network verification to good use. In *Proceedings of HotNets '19*, page 77–84. ACM, 2019.

[15] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of SIGCOMM '16*, pages 328–341. ACM, 2016.

[16] Ann Bednarz. Global microsoft cloud-service outage traced to rapid bgp router updates. https://www.networkworld.com/article/971873/global-microsoft-cloud-service-outage-traced-to-rapid-bgp-router-updates.html, accessed on April 4, 2024, 2023.

[17] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[18] Karl S Brace, Richard L Rudell, and Randal E Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45, 1991.

[19] Larry Brader, Howie Hilliker, and Alan Wills. *Testing for Continuous Delivery with Visual Studio 2012*. Microsoft, 2013.

[20] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *Proceedings of SIGCOMM '23*, page 122–135. ACM, 2023.

[21] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. Relational verification using reinforcement learning. In *OOPSLA*, 2019.

[22] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of SIGCOMM '14*, pages 443–454. ACM, 2014.

[23] Cisco Systems, Inc. Configure protocol redistribution for routers. https://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/8606-redist.html.

[24] Codecov. Codecov: The leading code coverage solution. https://about.codecov.io/, accessed on June 23, 2021, 2021.

[25] Christof Ebert and Panos Louridas. Generative ai for software practitioners. *IEEE Software*, 40(4):30–38, 2023.

[26] C. C. Elgot and J. E. Mezei. On relations defined by generalized finite automata. *IBM J. Res. Dev.*, 9(1):47–68, jan 1965.

[27] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *Proceedings of (OSDI 16*, pages 217–232. USENIX Association, 2016.

[28] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proceedings of NSDI 15*, pages 469–483. USENIX Association, 2015.

[29] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on networking*, 9(6):681–692, 2001.

[30] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of SIGCOMM '16*, pages 300–313. ACM, 2016.

[31] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. Nv: An intermediate language for verification of network control planes. In *Proceedings of PLDI '20*, pages 958–973. ACM, 2020.

[32] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 302–313, 2013.

[33] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of SIGCOMM '09*, page 51–62. ACM, 2009.

[34] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions On Networking*, 10(2):232–243, 2002.

[35] GNU Guix. lcov–code coverage tool that enhances gnu gcov. https://guix.gnu.org/en/packages/lcov-1.15/, accessed on May 26, 2022.

[36] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of SIGCOMM '15*, page 139–152. ACM, 2015.

[37] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[38] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *Proceedings of NSDI 17*, pages 735–749. USENIX Association, 2017.

[39] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200. IEEE, 1994.

[40] Istio. Diagnose your configuration with istioctl analyze. https://istio.io/latest/docs/ops/diagnostic-tools/istioctl-analyze/.

[41] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963. ACM, 2019.

[42] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *Proceedings of SIGCOMM '19*, pages 200–213. ACM, 2019.

[43] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[44] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proceedings of NSDI 13*, pages 99–111. USENIX Association, 2013.

[45] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of NSDI 12*, pages 113–126. USENIX Association, 2012.

[46] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of NSDI 13*, pages 15–27. USENIX Association, 2013.

[47] Petr Lapukhov, Ariff Premji, and Jon Mitchell. Use of bgp for routing in large-scale data centers. *Internet Requests for Comments RFC Editor RFC*, 7938, 2016.

[48] Krister Lindén, Miikka Silfverberg, and Tommi Pirinen. Hfst tools for morphology–an efficient open-source package for construction of morphological analyzers. In *State of the Art in Computational Morphology: Workshop on Systems and Frameworks for Computational Morphology, SFCM 2009, Zurich, Switzerland, September 4, 2009. Proceedings*, pages 28–47. Springer, 2009.

[49] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of SOSP '17*, page 599–613. ACM, 2017.

[50] Nuno P Lopes and Andrey Rybalchenko. Fast BGP simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 386–408. Springer, 2019.

[51] Matthew Luckie, Bradley Huffaker, Amogh Dhamdhere, Vasileios Giotsas, and KC Claffy. AS relationships, customer cones, and validation. In *Proceedings of IMC '13*, pages 243–256, 2013.

[52] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *Proceedings of SIGCOMM '11*, pages 290–301. ACM, 2011.

[53] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 3rd ed edition, 2012.

[54] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *Proceedings of NSDI 20*, pages 953–967. USENIX Association, 2020.

[55] Bruno Quoitin and Steve Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE network*, 19(6):12–19, 2005.

[56] Steve Ragan. BGP errors are to blame for Monday's Twitter outage, not DDoS attacks. https://www.csoonline.com/article/3138934/security/bgp-errors-are-to-blame-for-monday-s-twitter-outage-not-ddos-attacks.html, accessed on June 23, 2021, 2016.

[57] Deon Roberts. It's been a week and customers are still mad at BB&T. https://www.charlotteobserver.com/news/business/banking/article202616124.html, accessed on June 23, 2021, 2018.

[58] Deon Roberts. Facebook says its outage was caused by a cascade of errors. https://www.nytimes.com/2021/10/05/technology/facebook-outage-cause.html, accessed on Feb 23, 2022, 2021.

[59] Mike Robuck. Due to a router misconfiguration, cloudflare suffers short outage on friday. https://www.fiercetelecom.com/telecom/due-to-a-router-misconfiguration-cloudflare-suffers-short-outage-friday, accessed on Feb 23, 2022, 2020.

[60] Joao Luis Sobrinho. Network routing with path vector protocols: Theory and applications. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 49–60, 2003.

[61] Fabio Somenzi. CUDD: CU decision diagram package release 2.5.0. *University of Colorado at Boulder*, 2012.

[62] Alibaba Open Source. Rela. https://github.com/alibaba/rela, accessed on June 3, 2024, 2024.

[63] Tom Strickx and Jeremy Hartman. Cloudflare outage on june 21, 2022. https://blog.cloudflare.com/cloudflare-outage-on-june-21-2022, accessed on April 4, 2024, 2022.

[64] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context to prevent production failures. In *Proceedings of OSDI'20*. USENIX Association, 2020.

[65] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of SIGCOMM '16*, pages 426–439. ACM, 2016.

[66] Yevgeniy Sverdlik. Microsoft says config change caused azure outage. https://www.datacenterknowledge.com/archives/2014/11/20/microsoft-says-config-change-caused-azure-outage, accessed on Feb 23, 2022, 2014.

[67] Yevgeniy Sverdlik. United says it outage resolved, dozen flights canceled monday. https://www.datacenterknowledge.com/archives/2017/01/23/united-says-it-outage-resolved-dozen-flights-canceled-monday, accessed on June 23, 2021, 2017.

[68] Cisco Systems. Ip routing: Bgp configuration guide. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bgp/configuration/xe-16/irg-xe-16-book.html, accessed on April 4, 2024, 2016.

[69] Cisco Systems. Overview of netflow. https://www.cisco.com/c/dam/en/us/td/docs/routers/asr920/configuration/guide/netmgmt/fnf-xe-3e-asr920-book.html, accessed on Feb 2, 2024, 2021.

[70] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, jun 1968.

[71] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. Safely and automatically updating in-network ACL configurations with intent language. In *Proceedings of SIGCOMM '19*, page 214–226. ACM, 2019.

[72] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[73] Route Views. University of Oregon Route Views project. http://www.routeviews.org/routeviews/, 1997.

[74] Rosemary Wang. Testing HashiCorp Terraform. https://www.hashicorp.com/blog/testing-hashicorp-terraform.

[75] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *Proceedings of OOPSLA 2016*, pages 765–780. ACM, 2016.

[76] Zach Whittaker. T-mobile hit by phone calling, text message outage. https://techcrunch.com/2020/06/15/t-mobile-calling-outage/, accessed on June 23, 2021, 2020.

[77] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. *ACM SIGCOMM Computer Communication Review*, 44(4):383–394, 2014.

[78] Xieyang Xu, Ryan Beckett, Karthick Jayaraman, Ratul Mahajan, and David Walker. Test coverage metrics for the network. In *Proceedings of SIGCOMM '21*, page 775–787. ACM, 2021.

[79] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Trans. Netw.*, 24(2):887–900, April 2016.

[80] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. Accuracy, scalability, coverage: A practical configuration verifier on a global WAN. In *Proceedings of SIGCOMM '20*, page 599–614. ACM, 2020.

[81] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of CONEXT '12*, pages 241–252, 2012.

[82] 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 *Proceedings of NSDI 14*, pages 87–99. USENIX Association, 2014.

[83] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. Differential network analysis. In *Proceedings of NSDI 22*, pages 601–615. USENIX Association, 2022.

[84] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of SIGMOD '10*, pages 615–626. ACM, 2010.

Appendix A

EVALUATION DATASET OF Rela

A.1 Example changes and Rela specs

We present ten example changes, drawn from the dataset in §5.8, and their Rela specs. Various aspects of these specifications, such as locations and IP addresses, have been anonymized.

Change 1: Disabling certain internal forwarding paths. Network engineers want to disable forwarding paths inside the backbone network from region A to region B for certain prefixes, such that traffic to these prefixes will exit the backbone network via `exit2` in region A and take another path to region B, as requested by users. No other traffic should be affected. The implementation of the change modifies the configurations on the iBGP route reflectors in region B, such that the iBGP announcements from region B to region A with related prefixes will be denied by routing policies.

The intent of this change can be expressed in Rela as:

```
regex a := where(region=="A")
regex b := where(region=="B")
spec change1 := if (dst in 1.1.1.0/24) {
    a*      : preserve;
    b*exit1 : replace(.*, exit2);
    >>
    .*      : preserve;
} else {
    .*      : preserve;
```

```
}
```

The granularity of this spec is device group level.

Change 2: Expanding a device group. Network engineers want to expand a device group from n routers to $2n$ routers, such that traffic that traversed one or more pre-existing routers should now go through one or more routers in the new group. No other traffic should be affected.

The intent of this change can be expressed in Rela as:

```
regex dg_old := r_1|...|r_n
regex dg_new := r_1|...|r_2n
spec change2 := {
    .*      : preserve;
    dg_old  : any(dg_new);
    .*      : preserve;
    >>
    .*      : preserve;
}
```

The granularity of this spec is device level.

Change 3: Adding community tag to certain BGP prefixes. Network engineers want to add a new community tag to certain BGP routes. The implementation involves changing the export routing policy in the configurations of border routers. Despite the changes to routing policies, engineers want to ensure that no existing flow changes its paths at the device level.

The intent of this change can be expressed in Rela as:

```
spec change3 := .* : preserve;
```

The granularity of this spec is device level.

Change 4: Publishing a new internal IP prefix. Network engineers want to establish

connectivity from certain source datacenters (S1, *etc.*) to a particular destination datacenter (D) for a newly assigned IP prefix (e.g., 1.1.1.0/24).

The intent of this change can be expressed in Rela as:

```
regex s := where(datacenter=="S1") | ...
regex d := where(datacenter=="D")
spec change4 := if (dst in 1.1.1.0/24) {
    .* : any(s.*d);
} else {
    .* : preserve;
}
```

The granularity of this spec is device group level.

Change 5: Shifting traffic to new interconnection routers. Network engineers want to shift traffic that leaves three regions (R1, R2 and R3) from an old set of interconnection routers (i1, i2 and i3 respectively denotes the device group that serves this purpose in the three regions) to a new set of router groups (i1', i2' and i3'). The traffic between these three regions should be shifted from interconnection links such as i1-i2 to new links such as i1'-i2', while traffic to regions other than R1, R2 and R3 should exit via the same border routers as before. As always, traffic in other parts of the backbone should not be affected by this change.

The intent of this change can be expressed in Rela as:

```
spec change5 := {
    region1   : preserve;
    i1        : replace(i1, i1');
    border1.* : preserve;
    >>
    region1  : preserve;
    i1i2     : replace(i1i2, i1'i2');
```

```
    region2* : preserve;
    >>
    region1  : preserve;
    i1i3     : replace(i1i3, i1'i3');
    region3* : preserve;
    >>
    ...
    // rotate 1, 2, and 3
    ...
    >>
    .* : preserve;
}
```

The granularity of this spec is device group level.

Change 6: Shrinking edge links. Network engineers want to shrink the capacity of an edge link connected to an external peer. This is implemented by disconnecting some of the parallel links connected to this peer.

The intent of this change can be expressed in Rela as:

```
spec change6 := {
    .*         : preserve;
    exits_old : any(exits_new);
}
```

The granularity of this spec is interface level. But our evaluation uses a device-level version because the network simulation tool does not retain information about exit interfaces.

Change 7: Upgrading router firmware. Network engineers want to upgrade the firmware of a router. After the upgrade, the configuration syntax was updated by the manufacturer, and thus engineers updated the configuration content accordingly. In such cases, it is

critical to check that the forwarding behaviors are equivalent before and after the change.

```
spec change7 := .* : preserve;
```

The granularity of this spec can be at any level. (A finer-grained analysis is likely to catch more errors, but at a greater computational cost.)

Change 8: Shifting local traffic. Network engineers want to disable the link between a high-tier router group H1 and its low-tier neighbor router group L1, and they would like the H1-L1 traffic to detour via L1's sibling group L2, *i.e.*, H1-L2-L1.

The intent of this change can be expressed in Rela as:

```
spec change8 := {
    .*    : preserve;
    H1L1  : any(H1L2L1);
    .*    : preserve;
    >>
    .*    : preserve;
    L1H1  : any(L1L2H1);
    .*    : preserve;
    >>
    .*    : preserve;
}
```

The granularity of this spec is device level.

Change 9: Fixing IGP cost. Network engineers want to lower the IGP (interior gateway protocol) cost of the link between R1 and R2, such that traffic will go through direct R1-R2 link instead of detouring via other routers.

The intent of this change can be expressed in Rela as:

```
spec change9 := {
    .*      : preserve;
```

```
    R1.*R2 : replace(R1.*R2, R1R2);
    .*     : preserve;
    >>
    .*   : preserve;
}
```

The granularity of this spec is device level.

Change 10: Modifying topology. Network engineers aim to modify the topology such that a low-tier router group L is connected to mid-tier router group M. The router group L was originally connected to a high-tier router group H before the change.

The intent of this change can be expressed in Rela as:

```
spec change10 := {
    .*  : preserve;
    LHM : any(LM);
    .*  : preserve;
    >>
    .*  : preserve;
    LH  : any(LMH);
    .*  : preserve;
    >>
    .*  : preserve;
    MHL : any(ML);
    .*  : preserve;
    >>
    .*  : preserve;
    HL  : any(HML);
    .*  : preserve;
    >>
```

```
    .*  : preserve;
}
```

The granularity of this spec is device group level.

A.2 Performance Evaluation Dataset

We released a sample data plane generated by simulation of the same backbone network as studied in §5.8. The dataset contains 22,000 packet equivalent classes (PECs) together with the set of packets used for simulation, the before-change forwarding paths, and the after-change forwarding paths of each PEC. Router names, interface names, and IP addresses in the dataset have been anonymized for confidentiality. This dataset works out-of-the-box with the open-sourced Rela tool [62] and can be used to benchmark the performance of Rela or other similar tools. We note that this dataset contains only a small fraction of sampled PECs compared to the dataset used in §5.8. While it is sufficient to benchmark per-PEC verification performance, total verification cost grows in proportion to the total number of PECs in the change.