Network Control Plane Synthesis and Verification

Ryan Andrew Beckett

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: David Walker

September 2018

Abstract

Computer networks have become an integral part of modern infrastructure, and as the world continues to become increasingly interconnected and more devices come online, the importance of networks will only continue to grow. A critical component of networks is a process called routing, whereby the network determines how to move data from point A to point B as changes occur dynamically (*e.g.*, when new devices connect or equipment fails). Routing is traditionally achieved through the manual configuration of one or more distributed protocols that exchange messages about available routes to different destinations. Manual configuration lets a network operator tune various low-level protocol parameters to accommodate the different economic-, performance-, and robustness-related objectives that they may have for the network. Unfortunately, the low-level nature of existing configuration primitives and the scale of modern networks makes it difficult for humans to predict the impact of configuration on all possible runtime behaviors of the network, often resulting in configuration bugs.

This dissertation develops two complementary techniques for proactively finding and preventing bugs in configurations. The first technique is verification. Given a collection of router configurations and a high-level specification of what the network should do (*e.g.*, certain devices should be reachable), verification aims to ensure that the configurations implement this high-level specification correctly for all possible network behaviors. To address this problem, we develop a formal model of network routing protocols and show how many common protocols can be translated to logic constraints that existing constraint solvers can solve to find (or prove the absence of) bugs. The second technique is synthesis. Given a high-level specification of what the network should do, synthesis aims to produce a collection of configurations that faithfully implement the specification for all possible dynamic network conditions. We develop a new high-level language for describing end-to-end network behavior and demonstrate an efficient synthesis algorithm that can generate correct configurations. Throughout the development of both techniques, we show the importance of "abstraction" in speeding up each technique by several orders of magnitude.

Acknowledgements

I want to start by thanking my advisor David Walker for all his teaching, patience, and guidance over the years. Dave has made an otherwise arduous journey a delight. It has also been a pleasure to work with many great co-authors over the years, including Jitu Padhye, Ratul Mahajan, Todd Millstein, Sharad Malik, Aarti Gupta, Jennifer Rexford, Michael Greenberg, Shuyuan Zhang, Kelvin Zou, and Ang Chen – all of whom have taught me a great deal through our interactions. The members of the PL group at Princeton have also been a great source of support whether it was getting feedback on ideas and presentations or just taking my mind off of things. I hope I have been an equal source of support in return. I am grateful for the advice and attention from my dissertation committee: Aarti Gupta, Ratul Mahajan, Jennifer Rexford, Nick Feamster, and David Walker. Their feedback and efforts have greatly improved the quality of this thesis. This dissertation would also not have been possible without my undergraduate advisor Paul Reynolds, who introduced me to programming languages and opened my eyes to the possibility that graduate school might be something I would enjoy pursuing. I would like to thank my family – my mother, father, and my brother Matthew – who have been nothing but supportive over the years (even when I call for help with silly problems). Finally, much of the work in this dissertation was supported by NSF grants 1111520, 1525936, and 1703493 as well as funding from Cisco, Facebook, and Google.

Contents

List of Figures

Chapter 1

Introduction

Starting from its roots in the government-sponsored ARPANET project in the late 1960s, the Internet has seen explosive growth over the past half century that has propelled the technology from an obscure research experiment to a global piece of modern infrastructure that pervades nearly every aspect of modern life. The Internet ties together thousands of independently-operated private networks through a collection of common protocols that, collectively, enable reliable communication between devices anywhere in the world.

Key to communicating reliably between devices across a network is a process known as *routing*, whereby each device determines a path (or paths) through a network to use in order to reach a destination. When a device receives traffic for a destination, it consults a locally-computed routing table to determine which neighbor to send the packet to next. In practice, routing is complicated by the fact that there are many ways to deliver information across a network from point A to point B, with different characteristics and tradeoffs. For instance, one reasonable and simple way to route traffic is to always take the shortest path. However, different actors often have concerns other than distance travelled, such as security (e.g., keep the information inside the continental United States), performance (e.g., use a longer path with higher throughput), robustness (e.g., use a slightly longer path that is less prone to congestion), or cost-effectiveness (e.g., avoid high-cost neighbors).

Such flexible routing policy is made possible through *configuration*. Paths in routing are typically computed through one or more routing protocols that exchange information about different destinations in a purely distributed fashion. A human must author a separate configuration file for every device in the network, which tells each device how to process messages locally in different routing protocols. Humans try to write configurations such that, the emergent behavior of the routing computation achieves the security, performance, robustness, and cost-related goals for the network. Thus, routing in a network can be viewed as a massive distributed program written.

Unfortunately, humans are notoriously bad at writing, reasoning about, and maintaining distributed programs. To complicate matters further, existing languages for network configuration are extremely low-level, requiring authors to reason about specific protocol parameters at the level of device interfaces. In addition, different device vendors such as Cisco and Juniper have different configuration languages. For networks that use heterogeneous collections of devices, network operators must author configurations for different devices in different languages. A final complication is that, even if network operators can configure their networks correctly, networks are inherently dynamic in nature. Links and devices can (and often do) fail during network operation [71], and many networks must constantly evolve their policies to meet changing business demands.

Despite being designed many decades ago, such vendor configurations languages remain the primary way to implement network policy. If we were to compare network configuration languages to other general purpose languages, we would observe that this would be akin to having software developers write code in assembly language for each of several different architectures (e.g., x86 and ARM), describing operations on individual machine registers rather than program variables, but now for a distributed system where operations can fail arbitrarily.

An unfortunate consequence of the difficulty of network configuration is that policy violations and network outages are all-too-common, with major configuration-related outages making the news monthly [17, 68, 53]. For instance, a minor misconfiguration error in the Level 3 ISP backbone network in 2016 caused the largest network outage ever reported, with over 111 million phone calls dropped during the outage [88]. A less visible, but equally important problem with the

difficulty of configuration, is that network operators lack the agility needed to change and update their networks quickly to adapt to business requirements. As a result, networks can stagnate and become a bottleneck to new innovations.

The state-of-the-art for catching and fixing network misconfigurations in practice is to actively monitor certain network state at runtime and roll-back configurations if an issue is detected. While monitoring the network can often find when and where a problem is occurring, by the time a problem is found it has already impacted the actual network. Further, monitoring can only find configuration bugs that actively manifest themselves in the network, but there can be many latent bugs that remain hidden and can only be triggered in specific scenarios.

This thesis addresses the challenges of network configuration in two ways: The first is through configuration verification. Given a collection of configurations and a high-level specification of what the network should do, we can automatically check if the network will lead to the correct behavior for all possible inputs to the network (e.g., all packets, all environments, all failures, and so on). The second is through configuration synthesis. Rather than having a human write the configurations in the first place, we can automatically derive a set of configurations that correctly implement some high-level specification, which are again correct for all possible inputs to the network. We focus on these two approaches both because verification and synthesis provide extremely powerful guarantees about configuration correctness and because they are complimentary in nature (*e.g.*, verification can be used to check the output of a synthesis tool). However, both approaches are known to suffer from high-complexity. This means that for both approaches to configuration correctness, scaling to large networks is a challenge. To address this issue, this thesis explores a notion of *network abstraction* where a large, concrete network can be transformed into a smaller, abstract network in a way that is compatible with verification and synthesis.

In the remainder of this introduction, we start in Section 1.1 by giving an overview of the basics of network routing. In Section 1.2 we give examples and background information about network configuration. Section 1.3 and Section 1.4 cover the challenges and basics of performing verifica-

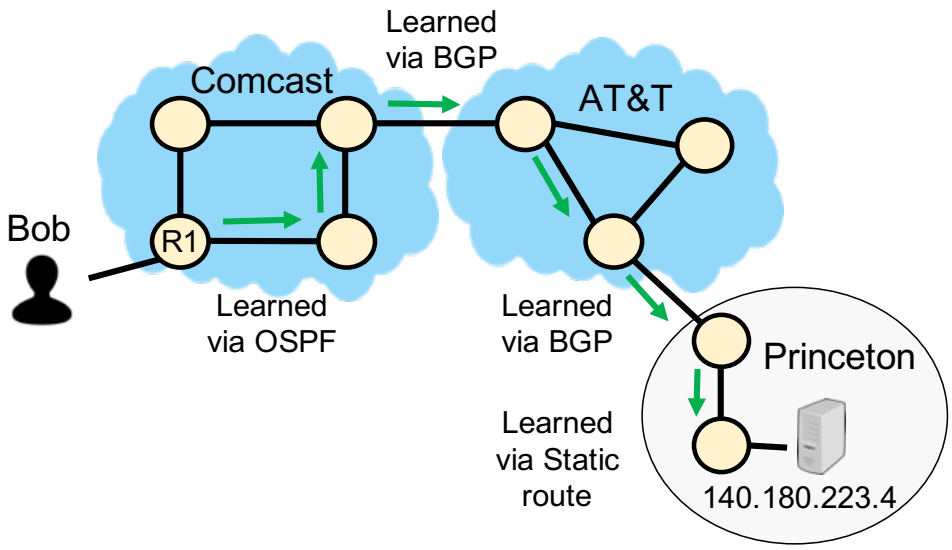


Figure 1.1: Routing in the Internet.

tion and synthesis in the context of distributed routing protocols. Finally, Section 1.5 describes the idea behind network abstraction in scaling verification and synthesis to large networks.

1.1 Network Routing

The Internet is comprised of thousands of independently-operated networks, called Autonomous Systems (ASes), each with their own routing objectives, goals, and priorities. These goals typically include local objectives such as, "ensure traffic can always reach my services" and "maximize network resilience and performance". Routing traffic through the Internet is achieved through the composition of routing performed for individual networks.

For instance, consider the example in Figure 1.1. if Bob wants to send traffic to a server hosted on Princeton's campus, Bob will send packets with a destination IP address for the server (`140.180.223.4`). Bob's ISP, Comcast, will have learned how to route packets with this destination IP through its neighboring network AT&T which Princeton uses for an ISP. AT&T can then deliver the traffic directly to Princeton, which in turn knows how to route the traffic to the server.

But how does this coordination for routing take place? Traditionally, each network runs multiple routing protocols, which communicate routes about destinations to neighbors (both internally and externally) in a distributed way. Collectively, we refer to this process of computing the routes to destinations as the *control plane*. However, different routing protocols have different characteristics and are often chosen for a variety of reasons. For example, ASes communicate routes using the Border Gateway Protocol (BGP). In the example, Princeton would advertise a BGP route to AT&T, which would advertise the route to Comcast. Comcast then uses this route to forward traffic for that destination (green arrows). Internally, traffic gets through Comcast's network to AT&T via the Open Shortest Path First (OSPF) protocol, which routes traffic based on the shortest cost path to the edge of Comcast's network. Similarly, AT&T might route traffic internally through its network using OSPF. Once Bob's traffic is inside Princeton's network, the traffic in this example is routed using a static route – a route that a human configures to always send traffic a certain direction.

Each routing advertisement communicates a route about a destination, which is represented as an IP *prefix*. In the example, the route for the server might be learned through the prefix `140.180.223.0/24`. A prefix can be thought of as representing a set of IP addresses. Here, the /24 means that the prefix contains any IP address whose first 24 bits are `140.180.223` and whose last 8 bits do not matter. Exchanging information about sets of packets at a time is important to avoid every router in the Internet having to know how to route to each of the 2^{32} IPv4 addresses.

Each router maintains a set of routes learned from different neighbors in a data structure called the Routing Information Base (RIB). A single RIB entry contains a destination prefix, a next-hop IP address, and any protocol-specific information. Consider the example network in Figure 1.2 with 4 routers R1-4, each running the BGP routing protocol. Bob wants to communicate with Alice, whose IP address is in the subnet `60.252.80.0/24`. This destination prefix is advertised by R4 into the BGP routing protocol to neighbors R2 and R3, which learn about the path to R4. R2 and R3 then advertise this route to R1, which learns about the path R2-R4 and R3-R4 to the destination. The RIB for R1 contains a single entry from each neighbor (R2 and R3). R1 decides which path to use based on a protocol-specific comparison of different fields. These fields can be

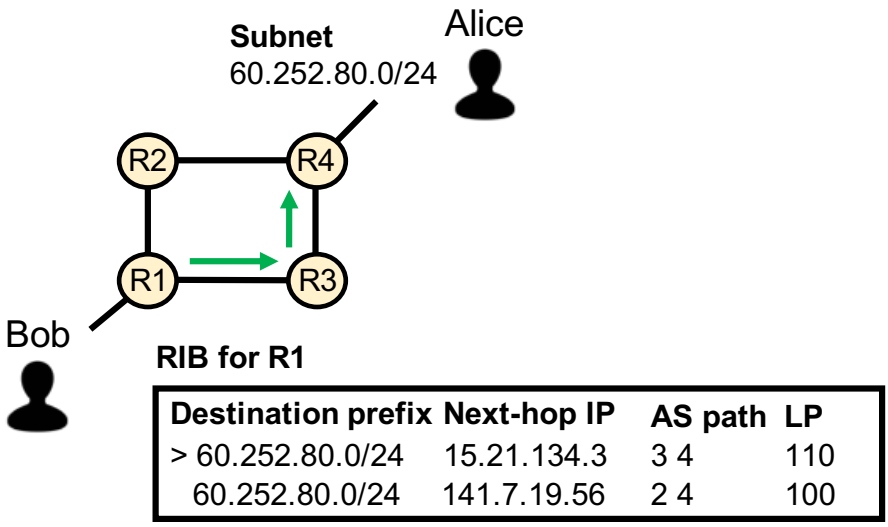


Figure 1.2: The Routing Information Base.

modified by configuration files. In the example, the (>) next to the first entry in the RIB for R1 is used because it has a higher BGP local-preference (LP) value for the path through R3 (local preference is a BGP-specific value that can be set by the operator to indicate a route preference). The RIB also contains the next-hop IP address that lets the router determine, locally, what port it should forward traffic for the destination out of.

Each router also keeps a secondary data structure called the Forwarding Information Base (FIB). Although each router stores a best route *per-neighbor* in the RIB, only the chosen routes (a single route per destination prefix) is stored in the FIB. These routes are then actually used to forward packets at runtime. When multiple FIB entries exist for the same destination IP address (e.g., `60.252.80.0/24` and `60.252.80.0/31` both contain `60.252.80.0`), then the route for the prefix with the longest prefix length will take priority (e.g., the /31). The FIB is designed to enable fast lookup in hardware.

1.2 Network Configuration

The decentralized nature of the Internet necessitates flexibility in routing. Different organizations have different business needs and preferences on how and where traffic should be sent to optimize for a number of requirements including cost, performance, reliability, and security. Business needs also often evolve over time as networks grow. The primary mechanism through which policy is achieved in routing is configuration.

1.2.1 How Configuration Influences Routing

While many details of routing protocols, such as the protocol message format, are fixed in advance, how routes are modified (or dropped) as they are passed between routers is typically defined by a human through configuration. For example, in the BGP protocol routes received from neighbors are first processed by an "import" filter before the router chooses a best route and exports this best route to neighbors after processing it through an "export" filter. Below is an example of a configuration file for a router that is written in Cisco's vendor-specific iOS format [1], for router R1 from the previous figure.

```
router bgp 1
  neighbor 172.0.2.1 remote-as 2
  neighbor 172.0.2.1 route-map IMPORT_R2 in
  neighbor 172.0.3.1 remote-as 3
  neighbor 172.0.3.1 route-map IMPORT_R3 in
  !
  route-map IMPORT_R2 permit 1
    match as-path 99
    set local-preference 100
  !
  route-map IMPORT_R3 permit 1
    match as-path 99
    set local-preference 110
  !
  ip as-path access-list 99 permit _4_
!
```

In this configuration, R1 is assigned the BGP Autonomous System (AS) number 1 (`router bgp 1`), has two neighbors R2 and R3, and declares an import filter for both neighbors (`IMPORT_R2` and `IMPORT_R3`). These filters examine protocol messages arriving at R1 from R2 and R3 and either drop or modify the messages. Both filters first match the AS-path (a list of AS numbers that records the path as a route is propagated in BGP), to see if it matches a regular expression (`_4_`) defined by the as-path list (named 99). The regular expression checks if the path has gone through AS 4 at some point. If so, then both import filters will modify the BGP local preference (to 100 if from R2, and to 110 if from R3). The resulting RIB would be that of Figure 1.2. The BGP protocol would then choose between the routes using a protocol-specific comparison of fields. For BGP, a higher local preference takes priority over all else. Hence, this configuration ensures that R1 will always prefer to use a route through R3 over one through R2.

Note that, in the example, even a simple configuration requires specifying a lot of low-level information (*e.g.*, interface IP addresses). Additionally, there are a large number of "magic" constants. For instance, the local preference of 100 vs 110 have specific meaning in BGP, and the name of configuration-level lists can clash (*e.g.*, 99) can clash with these other semantically-meaningful constants.

1.2.2 The Cost of Misconfiguration

As with any programming language, the flexibility in network configuration introduces the possibility for humans to introduce bugs when the low-level implementation diverges from their high-level mental model. Unfortunately, the aforementioned issues with configuration make such bugs quite common. This observation is bourne out empirically. For instance, the following is a small subset of the real incidents caused by network misconfiguration.

- In 2008, a Pakistani telecom accidentally advertised the wrong BGP prefix, which redirects traffic in the Internet from Youtube to Pakistan, creating a world-wide outage for YouTube [91].

- In 2012, a device misconfiguration in Microsoft's Azure network left the Azure Compute service unavailable to European customers for over 2 hours [95].
- In 2014, Time Warner customers experienced network outages due to misconfiguration in the Midwest [6].
- In 2015, a router misconfiguration caused United Airlines to ground more than 90 aircraft for over 2 hours, causing a widespread disruption [64].
- In 2016, Google engineers updated a router configuration, but due to human error, forgot to update other configurations, leading to dropped traffic. Their monitoring infrastructure caught the issue and attempted to roll back the configurations, but a bug in the roll-back software lead to a failure [90]. The outage lasted 46 minutes and affected the Google Compute Engine.
- In 2017, a misconfigured router in Level3's network led to a nation-wide outage for Comcast customers in the United States [37].

While these kinds of large, configuration-induced incidents are often widespread and publicly visible, less extreme misconfigurations are also quite common, with nearly half of all networks experiencing a configuration-related outage [87, 63] at some point. To make matters worse, the cost of such outages can be very high [63], ranging from thousands to even millions of dollars in lost revenue for every hour of downtime. In addition to issues stemming from network outages, another cost of misconfiguration is that of human time. Writing and testing configurations is a time-consuming process – a large network can easily have hundreds of thousands to millions of lines of configuration [41]. Like with any large software engineering project, even if written correctly initially, configurations are often slow to evolve as business demands change over time due to the high cost of maintaining a complex system.

Technique	Proactive	All packets?	All data planes?	Implementation Bugs?
Testing	Sometimes	No	No	Yes
Monitoring	No	No	No	Yes
Verification	Yes	Yes	Yes	No

Figure 1.3: Comparison of approaches to network correctness.

1.3 Verification of Configurations

Today, operators have a handful of tools for checking the correctness of their network. One common way to try to ensure robustness is through monitoring. The main downside of this approach is that monitoring will not proactively find bugs that exist in the network, and can only reveal issues after they have already happened. This is problematic as more services move into the cloud. For example, as cloud networks increasingly move towards service-level agreements (SLAs), even a couple of minutes of downtime can have a dramatic effect on their business.

Another popular approach is to use *testing*. Testing tools like traceroute [32] can inject a packet into a network and observe how it is forwarded. Testing can be done proactively if one can emulate or simulate the network offline. However, even so, testing will only check the correct behavior of a single packet in a data plane rather than checking the correctness of all packets. Because there can be 2^H different packets, where H is the number of bits in the packet header, testing all packets is intractable. Testing will also only check the correct behavior with respect to a single data plane – the current one, but in practice the state of the forwarding tables can change when any of a number of things happens: (1) a new BGP advertisement is received from a peer, (2) a link in the network fails, or (3) the order of messages changes during the execution of the routing protocol(s).

This thesis investigates an alternative approach, namely proactive *verification* of the network control plane (*i.e.*, the routing protocols running on the devices). Control plane verification is proactive – it can find bugs before configurations are deployed to the network. However, unlike testing, it is exhaustive. It can check a property like "traffic from A can reach B" for possible packets sent from A, and for all possible data planes that can emerge from the distributed routing computation. This requires building a mathematical model of the network control plane and then

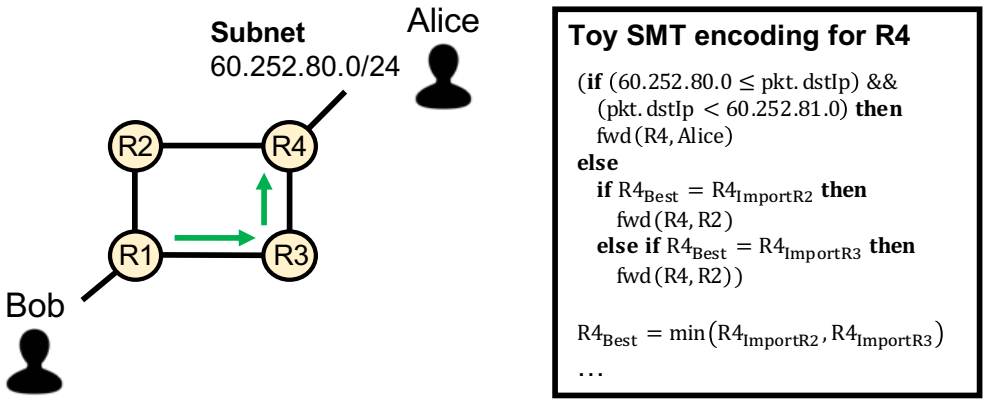


Figure 1.4: Example network encoding in SMT.

reasoning deductively about this model. However, one limitation of verification is that this "model" of the network may not be accurate with respect to the underlying implementation. (*e.g.*, if a vendor has an implementation bug). Approaches like monitoring that are not proactive, but instead measure the actual forwarding state of the network will catch implementation bugs that verification might miss. As such, the approaches should be considered complementary. Figure 1.3 summarizes the difference between these approaches.

The idea with verification in this dissertation is to encode a problem in a domain into a set of logic equations that describe all solutions to the routing problem. Efficient logic solvers such as Satisfiability Modulo Theory (SMT) solvers can quickly find if there exists an assignment to variables in the equations (i.e., inputs to the network such as link failures and peer advertisements) that satisfy the equations (e.g., that is a real behavior that can cause a property violation), or if no such assignment exists (i.e., the property always holds). As an example, consider the network SMT encoding show in Figure 1.4 for router R4. The encoding says, in logic, that if the packet destination IP is in the range defined by `60.252.80.0/24`, then R4 will forward traffic to Alice (since it has a directly connected interface for this prefix). Otherwise, if R4's best route is learned from R2, then it will forward to R2 and similarly for R3. Another logic constraint says that the best route learned at R4 is the minimum of those learned from R2 and R3, where minimum would also be defined for that protocol in logic.

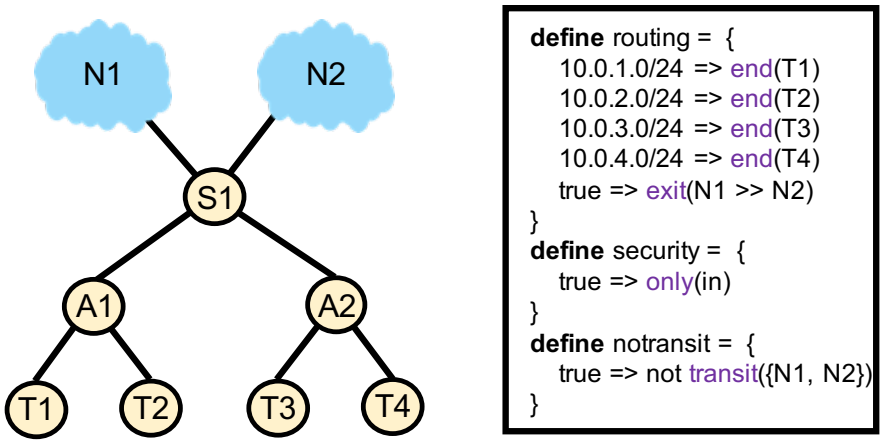


Figure 1.5: Example Propane policy.

1.4 Synthesis of Configurations

While verification can ensure that existing configurations are correct, it offers no guidance in coming up with the correct configurations in the first place. More specifically, given a collection of device configurations and a high-level property, verification can check that the configurations implement the high-level property. Synthesis turns this problem on its head and says instead: given a high-level property of the network, automatically derive configurations that are guaranteed to implement this property faithfully.

To enable synthesis of network configurations, we design and develop a new high-level routing language called Propane that allows users to define network policy at a high-level of abstraction. Propane policies describe the network behavior as a whole rather than device- and interface-specific behavior. In this way Propane is similar to using a higher-level programming language like C rather than programming directly in assembly. To bridge the gap between these levels of abstraction, the Propane compiler is responsible for turning these high-level programs into a distributed routing problem such that the outcome of the routing process faithfully implements the network-wide program. Figure 1.5 shows an example of a Propane policy for a toy data center network. At a high-level, the policy defines a collection of constraints (`define`). Each constraint contains a collection of pairs of a match on the type of traffic (destination IP) and a statement about

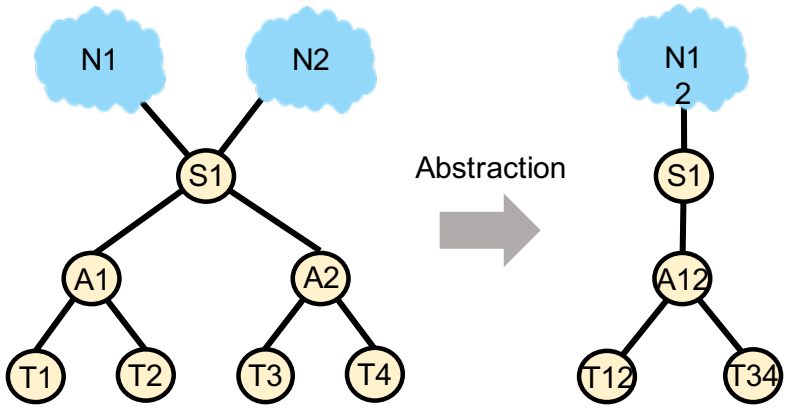


Figure 1.6: Example of a network abstraction.

the kinds of paths this traffic can use. For example, the first line of the `routing` constraint says that traffic destined for `10.0.1.0/24` must follow a path that ends at T1. The last line says that any traffic (`true`) should exit the network through either N1 or N2. The preference (>>) symbol indicates that a path through N1 should always be preferred, but a path through N2 can be used if no such path is available. The `security` constraint says that "all traffic should only traverse hops inside the data center", and the `notransit` constraint prevents traffic from using the data center as a transit point. The Propane compiler is responsible for combining, solving, and generating configurations that respect this high-level policy.

1.5 Abstraction: Scaling Network Analysis

Both verification and synthesis of network configurations are computationally expensive. For example, network verification of even a single fixed data plane produced from a routing problem with concrete inputs is an NP-complete problem [77]. At the same time, networks are only growing larger and more complex with economies of scale. Even today, modern data centers have expanded to include many thousands of routers [71].

To make the verification and synthesis techniques explored in this thesis scale to large networks, we investigate the application of *network abstraction* to the control plane. The idea in network abstraction is to find and exploit symmetries that exist in many networks to generate an

easier, but equivalent, problem to solve. Take for example, the network in Figure 1.6. The left hand side shows the original network, and the right hand side shows an abstracted network where several routers have been merged together. The goal of network abstraction is to determine when such consolidation is possible without affecting the outcome of the computation (i.e., for either verification or synthesis). By analyzing the smaller network only, both compilation and verification are often significantly faster.

1.6 Summary of Contributions

To summarize, this dissertation focuses on improving network reliability. It achieves this goal by addressing many of the issues caused by the difficulty of routing with network configurations through two complementary approaches: (1) verification of existing configurations, and (2) synthesis of new network configurations from a high-level language. Our contributions are as follows:

- **A formal model of the control plane.** In order to perform verification of network configurations, we must first develop a formal model of the network control plane. Our model, which we call the Stable Routing Problem (SRP), is based on classical work on the Stable Paths Problem (SPP) [56]. Like SPP, SRP models the paths to which the network will converge as a set of stability constraints. However SRP models this in terms of local conditions, which are then readily amenable to verification and generalizable to many different configuration protocols and features.

- **SMT-based verification of the control plane.** Using SRP as the foundation, we demonstrate how to translate a network using one or more routing protocols into an SMT encoding that captures all the possible data planes to which the network might converge under different inputs (e.g., failures, message orders, peer routes, etc.). In this way, we achieve a highly general approach to the verification of the network control plane.

- **A high-level language for routing.** We develop a new high-level language called Propane for describing the emergent routing behavior of a network. Propane is the first routing language that allows network operators to describe both intra-domain and inter-domain routing behavior by expressing end-to-end (rather than device-by-device) constraints on the the network control plane. Further, Propane is designed to express policy in a way that is protocol- and vendor-agnostic.

- **Synthesis of the control plane.** Given a Propane policy and a network topology, we describe an algorithm that bridges the gap between Propane and device-by-device configurations by synthesizing a set of network configurations running the distributed BGP protocol. The resulting configurations correctly implement the policy for *all* possible inputs to the network, including all possible link failures.

- **A theory of control plane abstraction.** For both verification and synthesis of the network control plane, we define (and prove sound) a theory of control plane abstraction that characterizes the types of network transformations that preserve correctness of the approach. By transforming and shrinking a network ahead of time, we are able to scale verification and synthesis to many large networks.

1.7 Additional Comments

Throughout this dissertation, there are several theorems pertaining to the correctness of the described approach to verification or synthesis. The full detailed proofs for all such theorems can be found in the appendix.

The work in this thesis is a revised and extended presentation of research developed through a series of co-authored papers [12, 13, 14, 15]. Chapter 3 on verification is based on a prior paper in SIGCOMM 2017 [12]. Chapter 4 on verification through abstraction is based on a followup paper in SIGCOMM 2018 [13]. Much of the material from Chapter 5 on synthesis is based on a paper

from SIGCOMM 2016 [14]. Finally, the majority of the material from Chapter 6 is based on work that appeared in PLDI 2017 [15].

Chapter 2

Background

In this chapter we give an overview of the basics of configuration-based routing, including a description of the most common distributed protocols and mechanisms used to compute the routes for forwarding traffic between devices, their typical use, advantages, and limitations in different network settings, and common configuration challenges that arise in practice with these protocols and mechanisms. Readers familiar with the basics of the network control plane such as thh RIB and FIB, as well as standard protocols such as RIP, OSPF, BGP (eBGP and iBGP), and static routes can skip ahead to Section 3.

2.1 Data Plane

The ultimate goal of routing is to compute a mapping from any packet entering a device on a particular interface to an outbound interface where the packet should leave the device. This mapping from inbound packet and interface to an outbound interface is referred to as the data plane. In practice, on real devices, the data plane is implemented using a data structure called the Forwarding Information Base (FIB). The FIB maintains a collection of entries, each of which contains a destination prefix and a next hop IP address. The destination prefix is used to classify groups of packets, and the next hop IP address determines the outbound port.

2.1.1 Longest Prefix Matching (LPM)

The choice of the FIB as a representation of the network data plane is useful for its compactness. Storing an entry for every single possible input packet is intractable since there can be 2^{32} such packets for IP version 4 (IPv4), and 2^{64} such packets for IPv6. By storing entries that contain a destination prefix rather than individual destination addresses, the data plane can capture the lookup behavior for entire sets of packets at a time. However, keeping entries for sets of packets can lead to forwarding ambiguity since different prefixes can contain overlapping sets of IP addresses. For example, the prefixes `60.252.80.0/24` and `60.252.80.0/31` both contain the IP address `60.252.80.0`. The router must decide which FIB entry to use when forwarding a packet. By convention, the prefix with the longest prefix length (*e.g.*, the /31 in the example) will be preferred since it contains more specific information. This choice together with the structure of the FIB lends itself nicely to efficient lookup on hardware using Ternary Content Addressable Memory (TCAM) [30]. TCAM memory is suitable for processing packets at network line rate because it can perform a parallel lookup to find the longest matching prefix, and hence the applicable FIB rule, directly in hardware.

2.1.2 Access Control Lists (ACLs)

While longest prefix matching provides a way to resolve forwarding ambiguity in the FIB, network operators may additionally want to perform security-based filtering on packets to determine if the packet should be allowed to pass through an interface. Access control lists, or ACLs for short, are a common mechanism used to enforce security policy in the network. ACLs provide a way to match and filter (drop or allow) certain packets that either enter or exit particular device interfaces. Notably, ACLs are a dataplane-only concept; ACLs have no effect on the control plane routing computation and are applied only after the routing process computes the routes that go into the FIB. Whereas routing determines how to forward a packet based on its destination IP address, ACLs provide much richer filtering capabilities. For instance one can match other fields of a packet such as its source IP address, source or destination port, protocol type, TCP flags, ICMP

code and so on. As an example, consider the following ACL, written in Cisco's iOS configuration language.

```
ip access-list extended HOST_OUT
  permit ip any 168.128.0.0 0.0.255.255
  permit tcp any any
  deny   ip any any
!
```

The first line defines a new ACL called `HOST_OUT`. The second line adds a rule to the ACL, which says that traffic to any destination IP, but with source IP address `168.128.0.0` and mask `0.0.255.255` should be allowed. The mask specifies the wildcard, or "don't care" bits, so this matches the same source addresses as the prefix `168.128.0.0/16`. The third line of the ACL allows any traffic for the TCP protocol with any source and destination IP. Finally, the last line applies to anything not matched by the first two lines, and drops (`deny`) any other packets.

2.2 Control Plane

The data plane enables fast forwarding of packets by mapping inbound packets on an interface to an outbound interface, but how this mapping is computed is determined by the control plane. Ideally, the control plane will compute FIBs for each device such that their combined forwarding behavior is able to transfer packets from their source to their correct destination along desirable paths. The control plane is typically implemented using one or more distributed protocols, with different tradeoffs. For example some protocols might converge faster, react to failures faster, use more or less CPU, or provide for more expressive user-defined configuration policies.

Broadly speaking, control plane routing can be broken down into two types of routing: (i) static routing, where routes that are determined at configuration time by a user, and (ii) dynamic routing protocols where routes are determined dynamically by communicating with peers and adjusting to network conditions such as failures.

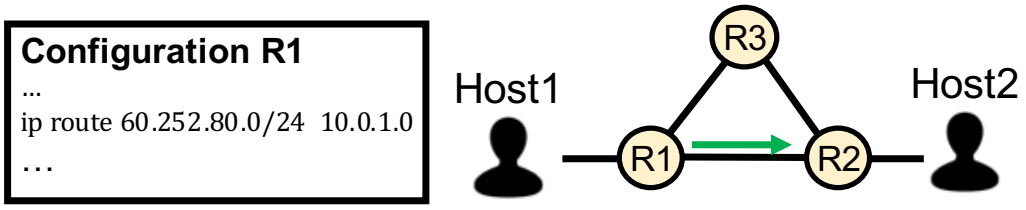


Figure 2.1: Example network using static routing.

2.2.1 Static Routing

One of the simplest ways to configure a network is with static routes. A static route is a configuration directive that says, for some destination prefix p, and router R, which next-hop neighbor should R forward the traffic to. In many instances, a device should always forward traffic a certain way (*e.g.*, if there is only one path), so there is no need to compute the route dynamically. Static routes are often useful in such situations for their simplicity (unlike dynamic protocols, they do not use any additional CPU), and their flexibility (the user can explicitly add any kind of forwarding behavior they want). However, static routing can be highly brittle because the routes will not adapt to changing network conditions. For example, when a link (or links) fail in the network that were used by a static route, the traffic using that static route will now be dropped by the router.

Consider the network in Figure 2.1. A static route on R1 is configured to always send traffic for prefix `60.252.80.0/24` directly to R2. If the R1 to R2 link fails, then Host1 will no longer be able to send traffic to Host2 despite the existence of a path through R3.

Another issue with static routes is that their extra flexibility lets users easily shoot themselves in the foot. For instance, if not configured carefully, static routes can lead to forwarding loops when interacting with other protocols. In the example, if R2 learned that the best route to the destination was through R1 (either through a static route or a dynamic protocol), then R1 and R2 would forward in a loop. In contrast, most dynamic protocols are designed to avoid forwarding loops by construction.

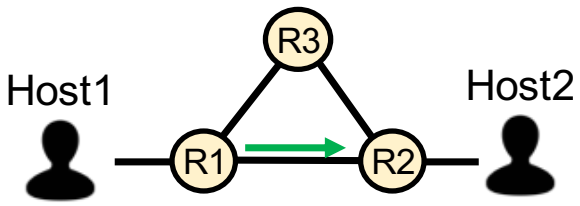


Figure 2.2: Example network using the RIP protocol.

2.2.2 Dynamic protocols

While static routing can be useful in certain situations, operators typically want to design a network that is robust to changing network conditions. There are many widely used protocols that can learn new routes and adapt to the network dynamically.

Routing Information Protocol (RIP): One such protocol is RIP. RIP is an old *distance-vector* protocol that routes packets along the path(s) with the shortest hop count to the destination. To achieve this, each router associates a "distance" to the destination from each neighbor in its RIB. Each neighbor has an initial distance of ∞. Periodically, routers will exchange their routing tables with each of their neighbors. They then compare each neighbor's routes with their own and update their current best route if a better route is available through that neighbor. RIP suffers from issues such as slow convergence time. As a result, the protocol caps the maximum hop count at 16 – any path longer than 16 hops is discarded. Figure 2.2 shows the same network as with static routing but running RIP. RIP would initially compute routes that result in the same forwarding behavior as before. However, now if the link between R1 and R2 fails, R1 will fall back to a route learned through R3 that has hop count 2.

Open Shortest Path First (OSPF): Another dynamic protocol that has become popular in practice is OSPF. OSPF is what is known as a *link-state* protocol. Topology information in the form of link-state advertisements (LSAs) about the state (up or down) of every link in the network is flooded to every router in the network. Each router builds what is called a link-state database

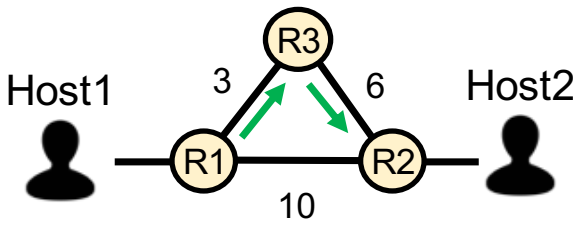


Figure 2.3: Example network using the OSPF protocol.

(LSDB). The routers then independently run Dijkstra's shortest path algorithm to compute the shortest path to each destination.

OSPF is designed to work with weighted links, which is useful for networks with asymmetry, where links may be connected to routers with varying geographic distance or have different capacity. Consider the network in Figure 2.3, which uses the same example as before, but with the OSPF-configured link weights shown. OSPF will compute routes such that Host1 will now forward to Host2 by going through R3 since this has the shortest cost path of 9. If the R3 to R2 link were to fail, each router running OSPF would recompute the shortest path to Host2 and R1 in particular, would now forward directly to R2.

Because OSPF requires flooding link-state information throughout the entire topology, its scalability is limited to small to medium sized networks, and is primarily used as an Interior Gateway Protocol (IGP) to provide connectivity between interfaces *inside* a single autonomous system. To make OSPF more scalable, the protocol allows users to split their network into multiple *areas* (groups of routers), where information between areas is kept in a summary form and propagated as in a distance vector protocol. Intra-area routes (routes learned inside an area) are preferred to inter-area routes (routes learned from another area).

Border Gateway Protocol (BGP): Protocols like RIP and OSPF serve as effective IGPs, providing connectivity to devices internally in a network. However, they have no way to learn or share routes with other independently operated networks. This task falls within the purview of the BGP protocol, which is used to provide connectivity between networks that lie under the control of different administrative entities. Because many organizations have different, and sometimes

competing, business requirements, BGP was designed in a way to allow for flexible and expressive routing policy.

BGP is a *path-vector* routing protocol, where the actual path that will be traversed is stored in each route advertisement. By storing the actual path, called the AS path, in routing advertisements, BGP prevents loops from forming, and allows for sophisticated policy decisions (*e.g.*, "avoid paths that leave the US").

When a route announcement is received by an AS running BGP, it is processed by custom import filters that may drop the announcement or modify some attributes. If multiple announcements for the same prefix survive import filters, the router selects the best one based on a lexicographic ranking of different attributes. This route is then advertised to neighbors, after being processed by neighbor-specific export filters that may drop the announcement or modify some attributes.

In addition to the AS-path, BGP advertisements contain a number of other attributes. One such attribute is a set of community strings. ASes use such strings to associate network-specific information with particular routes (*e.g.*, "entered on West Coast") and then use the information later in the routing process. Communities are also used to signal to neighbors how they should handle an announcement (*e.g.*, do not export it further). Another attribute is the multi-exit discriminator (MED). It is used when an AS has multiple links to a neighboring AS. Its (numeric) values signal to the neighbor how this AS prefers to receive traffic among those links.

All routes have a *local preference* attribute associated with them that can be set or modified in BGP filters. Routes with higher local preference are preferred. Among routes with the same local preference, other factors such as AS path length, MEDs, and internal routing cost, are considered in order. Because it is considered first during route selection, local preference is highly influential, and ASes may assign this preference based on any aspect of the route. A common practice is to assign it based on the commercial relationship with the neighbor. For instance, an AS may prefer in order customer ASes (which pay money), peer ASes (with free exchange of traffic), and provider ASes (which charge money for traffic). The combination of arbitrary import and export filters and route selection policies at individual routers gives BGP its flexibility.

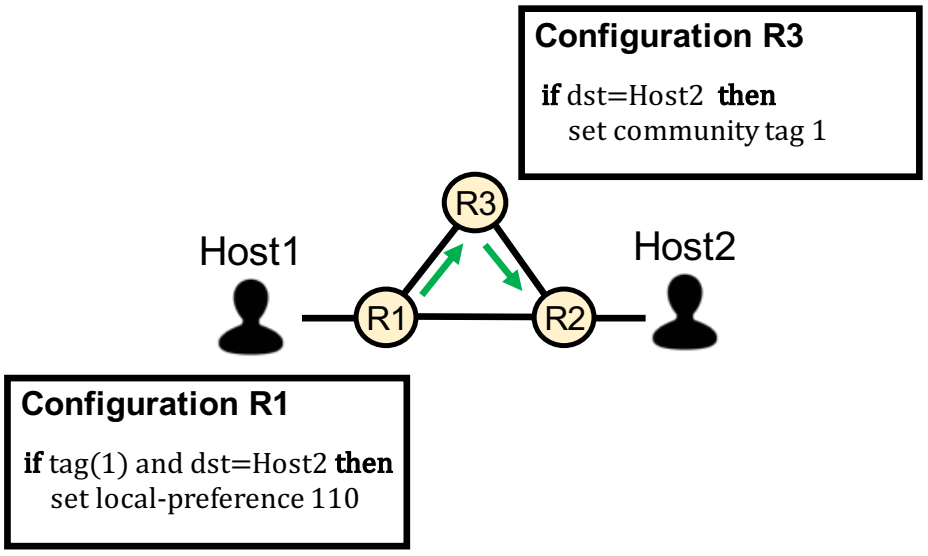


Figure 2.4: Example network using the BGP protocol.

BGP converges slowly, but is a highly-scalable protocol that can scale to handle the size of the Internet with hundreds of thousands of RIB entries (recall that a RIB entry is a route learned from a neighbor that consists of a destination prefix and some accompanying information based on the protocol). Although BGP was originally designed to provide connectivity between ASes, its flexibility and scalability has led to widespread use as an IGP as well (*e.g.*, it is widely used in data centers), where each router runs as its own AS [71]. The network in Figure 2.4 shows the same network from before now running the BGP protocol. Routers R1 and R3 have a BGP configuration written in pseudocode for simplicity. Each router is running BGP as its own AS. R3 has a configuration policy that checks if the destination prefix is for Host2, and if so, will add a community tag 1 to the route before exporting it to both R3 and R1. Similarly, R1 has a configuration (the same for each interface in this case) that will check for community tag 1, and if it is present, set the local-preference value to be 110. Because 110 is higher than the default value of 100, R1 will always prefer routes with this tag over those that do not have it.

A BGP route will originate from R2 and be sent to R1 and R3. R1 will learn about this direct route to R2 and use it. Likewise, R3 will receive the message, add the tag 1 to it, and send this message to R1. Finally, R1 will receive this message from R3, set the local preference to 110, and

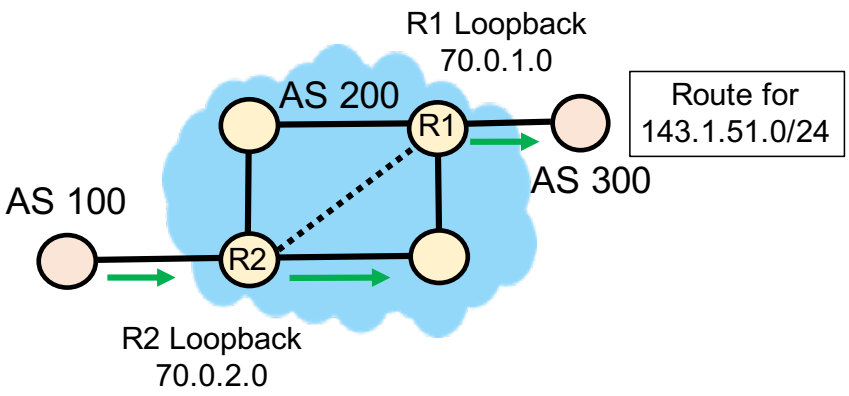


Figure 2.5: Example network using the iBGP protocol.

change to use the path through R3 instead. As a result, R1 will always prefer the top path when available. Like OSPF, R1 will switch to the direct path to R2 if the R3 to R2 or R1 to R3 links fail.

2.2.3 iBGP

While it is possible to run BGP with a single AS per router, a more common use case in wide-area networks is to use BGP hierarchically. An organization's network will appear to the outside world as a single AS and peer using BGP, while internally relying on an IGP, such as OSPF, to provide connectivity. An important feature of iBGP is that the IGP is not responsible for learning about destination prefixes that are not internal to the network (only the Loopback addresses – administrative addresses used to connect directly to a router). This is important from a scalability perspective as protocols like OSPF were not designed to handle the scale of the Internet. However, this layering of protocols introduces a number of complexities to make sure that the IGP and BGP protocols works together seamlessly.

The iBGP protocol bridges the gap between these different views of the world. Consider the following network running BGP in conjunction with OSPF. Figure 2.5 shows an example of a network using iBGP. iBGP works in the following way: First, all the border routers of the network are configured to run BGP (often called eBGP) in the usual way when peering with neighboring ASes. In the example, the border routers R1 and R2 run eBGP for network being configured

(AS200 in the blue cloud) and peer with ASes 100 and 300. The eBGP routers are then configured so that they are connected via iBGP edges. For instance R1 and R2 are connected by an iBGP edge (dashed line). These edges are virtual, and do not have to necessarily correspond to real edges in the network.

When a message from peer AS300 for prefix `143.51.1.0/24` arrives at border router R1, R1 accepts the route and initially chooses to forward through AS 300 (green line). R1 then exports the route to all of its BGP neighbors, including any iBGP neighbors – in this case just R2. But how can R1 send the route to R2 since they are not physically connected? It uses the IGP. In particular, the route for `143.51.1.0/24` is encapsulated in a new message that is destined for a special "Loopback" interface on R2, which an IGP like OSPF will typically be configured to learn to route to. For R1 however, the mechanism is simple – it just looks in its FIB to decide how to route to R2's Loopback address (`70.0.2.0`). R1 sends this new encapsulated message to R2, which then decapsulates this message to receive the route originally from AS 300. This route learned at R2 over iBGP is R2's best current route for `143.51.1.0/24`, so it decides to use this route. But how should R2 forward the packets for this destination since the R1–R2 edge does not exist in the actual topology? Once again, the Loopback address is used. Specifically, R2 will use a next-hop IP address for R1's Loopback address of `70.0.1.0`, which again an IGP like OSPF might know how to route. In the example, we assume OSPF will route to the bottom right router. Once again, the particular protocol is not important because R2 will determine the next hop for `70.0.1.0` by simply consulting its FIB for this destination IP. To prevent loops iBGP routes are never re-exported to other iBGP neighbors, so R2 will only export the route now to eBGP neighbors (in this case AS 100).

One can already observe that there is a problem. The router in the bottom right of the network will simply drop traffic because it has no route for `143.51.1.0/24`. It is for this reason that iBGP is typically run in a full mesh topology. The other two routers in this case could be configured to run iBGP with R1 and R2 as well, and thereby learn the route to the destination. Figure 2.6 shows the same network with the final forwarding behavior after iBGP is run in a full mesh.

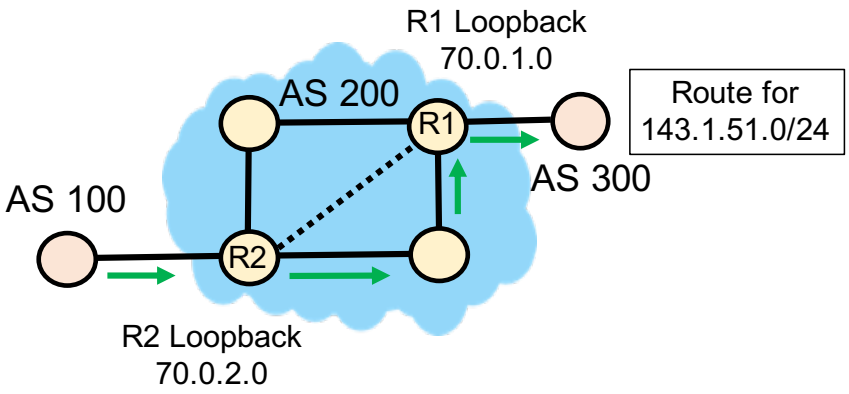


Figure 2.6: Example network from Figure 2.5 that is fixed.

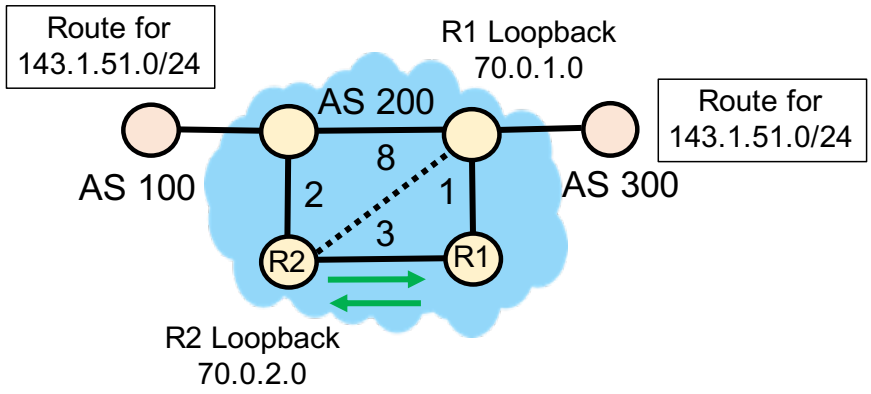


Figure 2.7: Route deflection in iBGP.

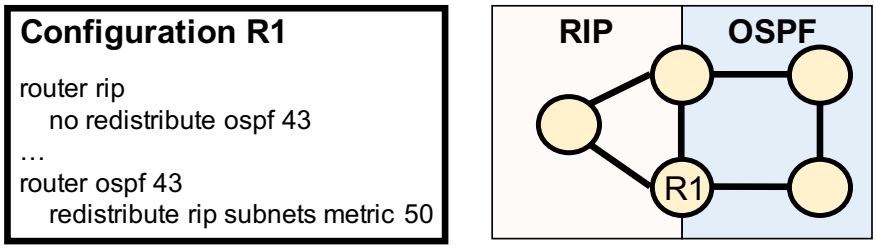
Example Bug: Another kind of bug that can occur in BGP is known as Route deflection [58]. Consider a slight modification of the example iBGP network shown in Figure 2.7. Here a route for `143.1.51.0` is advertised from both neighbors AS 100 and 300. Suppose that R1 and R2 set different BGP local preferences so that R1 prefers to go through AS 100 and R2 prefers to go through AS 300. Accordingly, both R1 and R2 will use the shortest IGP path to their exit point (AS 100 and 300 respectively). The shortest path for R1 goes through R2 and the shortest path for R2 goes through R1. At runtime, packets will thus be forwarded in a permanent loop. For this reason, ensuring consistent policy among iBGP peers is often considered best practice.

2.2.4 Route Reflectors

The full-mesh requirement for iBGP means that it cannot scale to large networks. Route reflectors help scalably disseminate iBGP information among BGP routers by acting as an intermediary. A designated few routers serve as "reflectors" for a collection of routers. Each reflector has a set of clients that it serves. When advertising a route to iBGP peers, a router will send the advertisement to its route reflector. This route reflector will then propagate the information to all other route reflectors, and each reflector then to all of its clients.

2.2.5 Route Redistribution

Route redistribution acts as a glue logic that lets various protocols with different formats interoperate and exchange information in a highly flexible, user-defined way. For example, in the network below, routes learned by R1 in the OSPF protocol can be configured to be redistributed into the RIP protocol so that RIP now knows about these routes and can advertise them to neighbors.

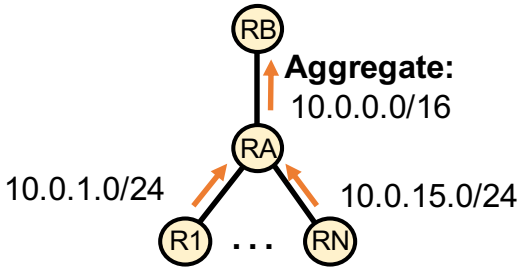


A problem with redistributing routes is that the message formats from different protocols typically contain incomparable "metrics". For example, OSPF uses cost based on weighted links while RIP uses the hop count. To ensure routes can be compared, when redistributing routes the user is required to specify a special "metric" value for the redistributed route, which is often called the Administrative Distance (AD). Routes with lower AD are preferred. In the example, routes redistributed from OSPF to RIP are given an AD of 50. The default AD for the RIP protocol is 120, so the OSPF routes will be preferred. The idea is similar to BGP's local preference mechanism but at the level of protocols. Also like BGP, redistributed routes can be filtered (dropped or modified) when passed between protocols. Naturally, naively injecting routes from one protocol into another

can lead to many complications and mistakes. For example, route redistribution can lead to routing loops and suboptimal routing [72].

2.2.6 Route Aggregation

As the size of the Internet grows, so too does the amount of memory needed to maintain routing tables. Routers participating in the BGP protocol can easily contain hundreds of thousands of routes learned from different ASes throughout the internet. [73]. This complexity has lead to the need for route "summarization", or *aggregation*. To minimize the size of routing tables, when announcing destinations to certain neighbors, a BGP router may instead announce a single prefix that covers multiple prefixes. For instance, in the example below, router RA is configured to advertise



`10.0.0.0/16` whenever it receives an advertisement from some subprefix like `10.0.1.0/24`. This saves router memory since upstream routers can keep a single prefix (`10.0.0.0/16`) instead of each of potentially many subprefixes (*e.g.*, `10.0.[0--15].0/24`).

While aggregation helps reduce the memory consumption of routers, it can lead to unexpected routing behavior such as traffic black holes [73]. For example, a router that announces `10.1.0.0/16`, because it has a route to `10.1.1.0/24`, may also get traffic for `10.1.2.0/24` to which it has no route.

2.2.7 Multipath Routing

Routing all traffic between two devices along the same path can lead to poor network utilization if certain devices send and receive more traffic than others. To better utilize network resources, mul-

tipath routing is often used to split traffic across multiple equally-good paths. However, sending packets that are part of the same flow – the same collection of header fields like source and destination IP address – along different paths can lead to out-of-order packet delivery, which results in very poor performance due to packet retransmissions. Instead, networks typically use Equal Cost Multipath Routing (ECMP), which pins flows to a paths by deterministically hashing the packet header to one output port out of several possible output ports. Thus, each packet is still mapped from an input port to an output port, but traffic may be split more uniformly throughout the network.

Chapter 3

Control Plane Verification

As we have seen, configuring networks correctly is challenging both due to the scale and complexity of the task. Many of the configuration primitives introduced in Chapter 2, if used incorrectly, can lead to unexpected consequences. Figure 3.1 provides a sample of several types of common misconfigurations that can arise in practice and their possible consequences. In this chapter, we approach the problem of misconfiguration through the lens of verification: Given a set of configurations and a high-level property P, check if the network will always satisfy P. We start by giving an overview on previous work related to network verification and how it fits in with this thesis in Section 3.1. In Section 3.4, we formally define a mathematical model of the network control plane as a collection of "stable" logical constraints, and we show how this model can be used for the purposes of network verification by leveraging SMT solvers in Sections 3.5 and 3.6. We conclude with a thorough evaluation of our implementation of a verification tool in Section 3.10.

3.1 Related Work

Over the years, researchers have developed many tools for finding errors in network configurations. We broadly divide these approaches into two classes of tools.

Misconfiguration	Possible Consequence
Bad ACL rule	Traffic is blackholed
Duplicate IP address	Unintentional anycasting of destination
Duplicate Loopback address	No iBGP connectivity
iBGP configured without Loopback	One failure breaks iBGP connectivity
iBGP not in a full mesh	Traffic is blackholed
Bad static route	Forwarding loops
Bad AD set in redistribution	Loops, Oscillation, Suboptimal routing
Splitting OSPF Areas poorly	Suboptimal routing
Misconfigured BGP MED	Violation of peering contract
Misconfigured BGP LP	Oscillation, More expensive provider
Inconsistent BGP export filters	Security breach, Carry traffic for free
Inconsistent BGP import filters	Cold-potatoes [42], Route deflection
Incomplete BGP export filters	Leaking private prefixes to the Internet
Incomplete BGP import filters	Peers "hijack" routes for internal prefixes
Allowing transit routes between peers	Network carries peer traffic for free
Misconfigured BGP AS path prepending	Route dropped from path overflow
Misconfigured Route Reflectors	Loops, Route deflection
Misplaced aggregation	Aggregation-induced blackhole

Figure 3.1: A sample of some types of misconfiguration and their consequences.

3.1.1 Analysis without formal semantic models.

One approach to network analysis is to use heuristics rather than a formal semantic model to find common types of mistakes. For instance, rcc [41] and other commercial products can find common mistakes and inconsistencies in configurations by checking the configurations against a collection of best practices and/or syntactic patterns. Another tool, Minerals [9], can find possible mistakes using machine learning to find similar and dissimilar configurations. While this type of approach can find a range of configuration errors, because it does not actually build a model of the network, it can report both false positives and false negatives and cannot answer questions about specific network behaviors.

3.1.2 Analysis with formal semantic models.

This thesis is primarily concerned with network configuration analysis based on a formal semantic model of the network. Analysis tools based on network models can be further classified according to their "coverage" along two dimensions:

- *Network design coverage*: how many types of network topologies, routing protocols, and other features that the tools supports; and
- *Data plane coverage*: how many (or how much) of the possible data planes that may arise in the network the tool can analyze. The network control plane dynamically generates different data planes as its environment (*i.e.*, up/down status of links and routing announcements received from external neighbors) changes. Tools with higher data plane coverage can analyze more possible data planes.

Dataplane analysis tools: Some of the earliest network diagnostic tools such as traceroute [32] and ping [31] can help find configuration errors by analyzing whether and how a given packet reaches its destination. These tools are simple but have high network design coverage—they can analyze forwarding for any network topology with a data plane computed using any routing protocol. However, they have poor data plane coverage—for each run, these tools analyze the forwarding behavior for only a single packet and only for the data plane that is currently installed in the network.

A more recent class of *data plane analysis* tools such as HSA [67] and Veriflow [69] have better data plane coverage. They can analyze reachability for all packets between two machines, rather than just one packet. However, the data plane coverage of such tools is still far less than ideal because they analyze only the data plane that is currently installed in the network. That is, they can only find errors after the network has produced the erroneous data plane.

Control plane analysis tools: *Control plane analysis* tools such as Batfish [44] can find configuration errors proactively, before deploying potentially buggy configurations. Batfish takes the

network configuration (*i.e.*, its control plane) and a specific environment (*e.g.*, a link-failure scenario) as input and analyzes the resulting data plane. This ability allows operators to go beyond the current data plane and analyze future data planes that may arise under different environments. Still, each run of Batfish allows users to explore at most one data plane, and given the large number of possible environments, it is intractable to guarantee correctness for all possible data planes. Most recently, several control plane analysis tools have gone from *testing* individual data planes to *verification*—that is, reasoning about many or all data planes that can be generated by the control plane. However, each such tool trades network design coverage for higher data plane coverage. For instance, while Bagpipe [100] can symbolically simulate the message-passing semantics of BGP in all possible environments, it assumes that the network is a single autonomous system (AS) connected in an iBGP full mesh, and does not model any internal routing. Another tool, ARC [49], translates configurations to a weighted graph where the weighted-shortest paths capture the network forwarding behavior. A single run of ARC can efficiently analyze multiple data planes by considering the consequences of all possible failures but not all possible sets of external routing messages. Further, many networks, such as those using iBGP or using certain features such as BGP local preference can not be reduced to simple weighted graphs. ERA [40] compactly represents a concrete set of control plane messages using binary decision diagrams (BDDs) and propagates this set along a path through the network by transforming the set as dictated by the network configuration. In this way, ERA can efficiently check reachability in certain large symbolic environments (*e.g.*, the environment with all possible eBGP advertisements), but using ERA to verify configurations in the face of all environments is an open problem [40]. Further, the path-based approach of ERA cannot faithfully analyze reachability for certain networks such as those running iBGP.

Summary and contributions: In summary then, while there has been great progress toward analyzing network configurations, no previous work has been able to answer the following question:

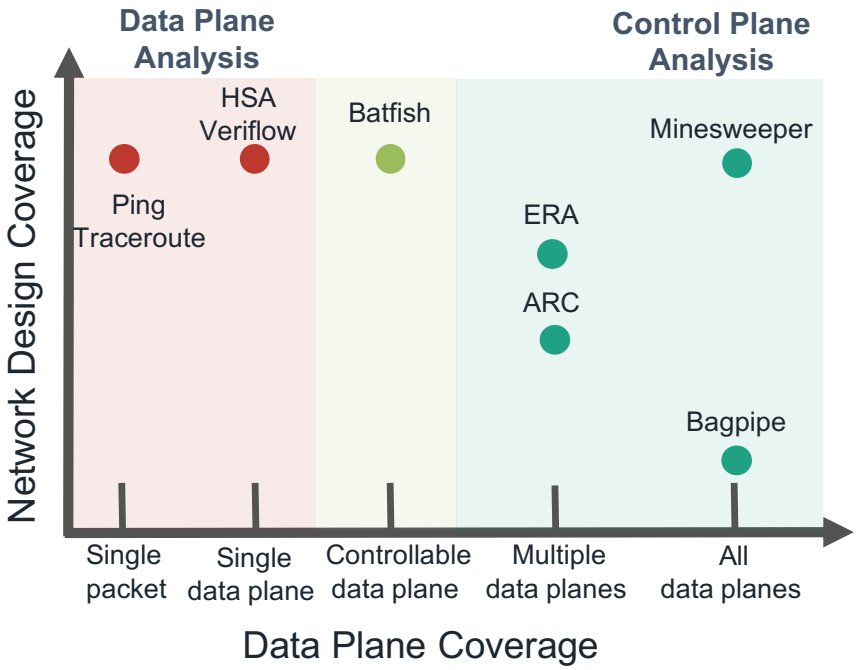


Figure 3.2: Landscape of network analysis tools.

Is it possible to build a verification tool that achieves both high network design coverage and high data plane coverage while remaining scalable enough to enable verification of many real networks?

This thesis demonstrates that this is possible by describing the theory and implementation behind a configuration verification tool called Minesweeper. Figure 3.2 situates Minesweeper and prior tools with respect to network design and data plane coverage. Minesweeper has both high network design coverage in that it works for a large collection of network protocols, features and topologies as well as high data plane coverage in that it can verify a large number of properties for all possible data planes that might emerge from the control plane. In the remainder of this chapter, we will describe the theory and implementation underlying Minesweeper.

3.2 Overview of the approach

The main challenges in developing Minesweeper were to: (1) develop a sufficiently general theory to model commonly used configuration primitives, and (2) scale such a general tool sufficiently to be able to reason about real networks. We addressed these two challenges by combining the following ideas from networking and verification literature:

Graphs (not paths): Most existing verification tools reason about individual network paths, for example by symbolically simulating a message traversing the path. While this approach has proven effective for stateless data plane analysis (*e.g.*, HSA [67]), it creates substantial problems for control plane analysis. The distinction is that, in stateless data planes, packets on one path never interfere with those on a different path; but in the control plane, two route announcements can interfere. A routing message along one path may be less preferred than a message over another path, causing it to be dropped when the other message is present. For accuracy, interactions along all paths must be modeled, but there can be an intractably large number of paths. We avoid this problem by using a graph-based model, where rich logical constraints on its edges and nodes encode all possible interactions of route messages.

In addition to its better accuracy, our model can verify a richer set of properties, expressed over graphs, rather than individual paths. For example, it can reason about equivalence of routers, load balancing, disjointedness of routing paths, and if multiple paths to the same destination have equal lengths. Such properties are often impossible for path-based models to check, and we show that they are valuable in finding bugs in real configurations.

Combinational search (not message set computation): Existing tools that analyze multiple environments [40, 100] eagerly compute the sets of routing messages that can reach various points in the network. However, these full sets are not typically needed and computing them is expensive. Fortunately, the symbolic model checking community has encountered this type of problem before. Rather than iteratively computing sets of messages, one can instead ask for a satisfying

assignment to a logical formula that represents all possible message interactions. Suppose a variable $x_{m,l}$ represents whether a message m reaches a location l in the network and N encodes the network semantics logically. If there exists a satisfying assignment to the formula $N \wedge x_{m,l}$=$true$, then m can reach l and all the constraints N imposed by interacting messages are also satisfied. The advantage of this formula-based approach is that while model checking with message set computation is PSPACE-complete [26, 92], the search for a satisfying assignment in the related model checking problem [18] is NP-complete. The intuition behind lower complexity is that searching for a satisfying assignment avoids computing many intermediate message sets. In practice too, modern SAT [79] and SMT (Satisfiability Modulo Theories) [36] solvers routinely solve large instances of such combinational search problems in hardware and software verification.

Stable routing problem: To realize an approach based on graphs and combinational search, we need to convert the distributed message-passing of the control plane into an equivalent logical formula. To do this, we define the Stable Routing Problem (SRP), a generic model of a routing protocol and the network on which it runs. SRPs can model networks running a wide variety of protocols including distance-vector, link-state, and path-vector protocols. SRPs are directly inspired by the stable paths problem (SPP) [56], but rather than describing the protocols' final solution using end-to-end paths (as SPPs do), SRPs describe runtime routing behavior in terms of local processing of routing messages, as configurations do. In addition to modeling configurations in terms of local constraints, this distinction allows SRPs to capture a wider variety of routing behaviors that emerge at runtime, such as forwarding loops. SRPs are also very similar to routing algebras [57, 93], but while routing algebras are primarily used to reason about properties of routing protocols (*e.g.*, convergence), we use SRPs primarily to reason logically about properties of routing protocol *instances* (i.e., a protocol running on a particular topology). Consequently, rather than encoding message exchanges, we can encode the solution to an SRP as a set of constraints on a graph directly in logic whose solutions can be found by an off-the-shelf SMT solver.

Slicing and hoisting optimizations: A naive encoding of an SRP into logic produces large formulas that cannot be solved quickly for real networks. We have designed a range of highly effective optimizations that reduce the number of variables and constraints in our generated formulae enormously. One class of optimizations is *slicing*, which analyzes the formula to remove variables and constraints that cannot affect the final outcome. A second class of optimizations is *hoisting*, which lifts repeated computations out of their logical context and precomputes them once. Intuitively, such optimizations are effective because real networks have simpler control planes than the theoretical worst case. For instance, in theory, messages can be arbitrarily modified when sent to neighbors (implying the need for different variables for messages to different neighbors), but in practice the same message is sent to multiple neighbors (allowing shared variables). Similarly, while different routers may have arbitrarily different control plane logic in theory, in practice many routers share parts of their configurations.

Implementation: We implement the concepts above in Minesweeper, and apply it to many real and synthetic networks. Across the 152 small- and medium-sized networks that we analyzed for four properties, we found 120 violations of the properties. One class of violations poses a serious security threat: the management interface IP of the routers could be "hijacked" by external neighbors by sending specific routing announcements. Our experiments with synthetic networks show that Minesweeper can verify rich properties such as many-to-one reachability, bounded path length, and device equivalence in under 5 minutes on networks with 100s of routers. Our optimizations are key to this performance. They help reduce verification time by a factor of up to 460x for large networks.

3.3 Motivating Example

Our approach represents two significant departures from existing work on configuration analysis: i) modeling the network as a set of constraints based on an SRP for the network graph, instead of

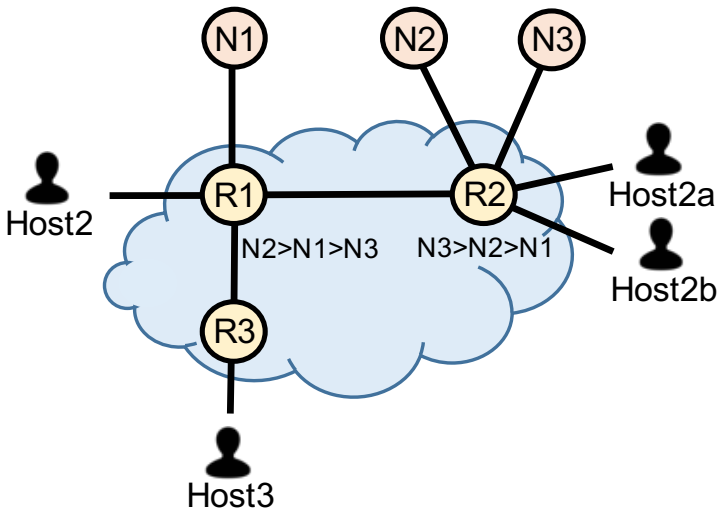


Figure 3.3: Path-based analysis can be unsound.

reasoning about source-destination paths; and $ii)$ using combinational search, instead of eagerly computing message sets. This section provides intuition behind these choices through an example.

Paths vs. graphs: Consider the network in Figure 3.3. It has three internal routers, R1 to R3, that we will assume run OSPF. It also connects to three external neighbors, N1 to N3, via BGP. The internal routers are connected to hosts, 1 to 3, whose address prefixes they redistribute into OSPF and BGP. R1 and R2 connect via iBGP, to share the BGP routes they hear from N[1..3]. They also redistribute BGP destinations into OSPF, so that R3 can reach those destinations, and OSPF into BGP so that internal subnets are announced externally. The BGP preferences of R1 and R2 are as shown: R1 (R2) prefers routes through N2, N1, and N3 (N3, N2, N1) in that order. Recall that in BGP, when multiple routes are available to the same destination, a router will select and share the most preferred one according to the local configuration.

Suppose we want to ensure that host3 uses N1 to reach any external destination even when all three of N1, N2 and N3 announce a path to that destination. Does this property hold in our network? The correct answer is positive, but interestingly, the answer a configuration analysis tool delivers depends on the sophistication with which it reasons about the interactions of control plane messages on different paths.

- If the analysis only considers the path N1-R1-R3, it will conclude that the property holds. R1 will select the route through N1 since no other route is available and pass it to R3. Thus, R3 (and host3) will send traffic through N1. (Data flows in the opposite direction to routing information.)

- If the analysis additionally considers the routing path N2-R2-R1-R3 (which interferes with the first path at router R1), it will conclude that the property does *not* hold. R1 will select the route through N2 and thus the route through N1 will not reach R3.

- If the analysis also considers N3-R2-R1-R3 (which interferes with the second path at R2 and the first path at R1), it will conclude once again that the property holds. R2 will select the route through N3, and thus R1 will select and propagate to R3 the route through N1.

In the general case, all possible paths can interfere with one another, and for correct analysis, all mutual interactions should be considered. But the number of paths can be enormous: $O(V^{\frac{E}{V}})$, where V and E are the number of nodes and edges (and thus $\frac{E}{V}$ is the average node degree). Existing path-based tools circumvent this problem by restricting the networks they can analyze (*e.g.*, Bagpipe [100]) or conducting a potentially unsound analysis (*e.g.*, ERA [40]). Our model avoids this problem by constructing a compact representation for all possible paths—a graph-based SRP encoding. The complexity of this structure is $O(V + E)$. Our graph accurately (and symbolically) models all interactions between different paths and supports a richer set of properties (described later).

Message sets vs. combinational search: One possible approach to control plane verification is to simulate all possible outcomes of the distributed control plane computation by computing (symbolic) sets of messages for all destinations. Once all outcomes of the control plane computation have been computed, one can analyze the complete set of possible final states and judge if the property of interest holds. Unfortunately, this approach often leads to a lot of unnecessary work.

In many cases, computing a full solution to the control plane computation is unnecessary as the validity of the property may not depend upon parts of that solution. In contrast, our approach

encodes both the network and the property in question as a logical formula. As an SMT solver searches for a satisfying assignment to the formula, it will take the property into account. If the property does not require knowledge of some aspects of the control plane, the search process may ignore that part of the model. For example, if R3 had an ACL that drops traffic sent to R1, then the solver may quickly learn that host3 can not reach N1 without reasoning about the full control plane behavior. In Section 3.10, we show that many properties can be checked much more efficiently for this reason.

However, approaches that compute message sets represent and store *all* possible outcomes of the control plane's full fixed point computation will find *all* violations of the property. In contrast, our approach *searches* for just *one* outcome of the control plane computation that violates the given property. The latter can be done extremely efficiently by modern SMT solvers in many domains. While our approach will not find *all* violations at once, finding just one violation can help pinpoint a bug. When that bug has been fixed, one can apply the procedure again.

3.4 Stable Routing Problem

We define the Stable Routing Problem – a formal model of the network control plane that we use for verification. An SRP is, in essence, a compact, parameterized description of a control plane protocol and the network graph on which it runs. Given an SRP, we can define a collection of logical constraints that characterize all the possible final routing behaviors that the network can exhibit after the network has converged. In the remainder of this section, we first define SRPs formally and then outline how they can model common routing protocols.

3.4.1 SRP Definition

An SRP is defined with respect to a single abstract notion of a routing "destination" (*e.g.*, a range of destination IP addresses). Since each destination can have its own routing behavior, we fix a single destination for each SRP instance. As shown in Figure 3.4, an SRP instance is a tuple

SRP instance $\boxed{SRP = (G, A, a_{\mathrm{d}}, \prec, \mathsf{trans})}$

$$
\begin{array}{lcll}
G & = & (V, E, d) & \textit{network topology} \\
V & & & \textit{topology vertices} \\
E & : & V \times V & \textit{topology edges} \\
d & : & V & \textit{destination vertex} \\
A & = & A' \cup \{\bot\} & \textit{routing attributes} \\
a_{\mathrm{d}} & : & A & \textit{initial route} \\
\prec & \subseteq & A \times A & \textit{comparison relation} \\
\mathsf{trans} & : & E \times A \to A & \textit{transfer function}
\end{array}
$$

SRP solution $\boxed{\mathcal{L} : V \to A}$

$$
\mathcal{L}(u) = \begin{cases} a_{\mathrm{d}} & u = d \\ \bot & \mathsf{attrs}_{\mathcal{L}}(u) = \emptyset \\ a \in \mathsf{attrs}_{\mathcal{L}}(u) \text{ that is minimal by } (\prec), & \mathsf{attrs}_{\mathcal{L}}(u) \neq \emptyset \end{cases}
$$

$$
\begin{array}{l}
\mathsf{attrs}_{\mathcal{L}}(u) \quad = \{a \mid (e, a) \in \mathsf{choices}_{\mathcal{L}}(u)\} \\
\mathsf{choices}_{\mathcal{L}}(u) = \{(e, a) \mid e = (u, v),\ a = \mathsf{trans}(e, \mathcal{L}(v)), a \neq \bot\} \\
\mathsf{fwd}_{\mathcal{L}}(u) \quad = \{e \mid (e, a) \in \mathsf{choices}_{\mathcal{L}}(u),\ a \approx \mathcal{L}(u)\}
\end{array}
$$

$$
a_1 \approx a_2 \iff a_1 \not\prec a_2 \wedge a_2 \not\prec a_1
$$

Figure 3.4: Formal definition of an SRP and its solutions.

$(G, A, a_{\mathrm{d}}, \prec, \mathsf{trans})$. The network topology is represented as a graph $G = (V, E, d)$ with a set of vertices V, a set of directed edges $E : V \times V$, and a destination vertex $d \in V$. The set A represents *attributes* that describe the format of routing messages. For example, A might be natural numbers representing path cost for OSPF, or when modeling BGP, A might represent tuples of a 32-bit local-preference value, a set of 16-bit community values, and a list of ASes representing the AS path and so on. For any SRP, we add a special value $\bot$ to represent the absence of an attribute. This is useful to model, for example, when a router has no route to a destination. We use a special attribute value a_{d} to represent the initial protocol message advertised by the destination d. For example, in OSPF this might be a path with cost 0, or in BGP this might be a BGP advertisement with the empty AS path.

In the SRP instance, $\prec$ is a partial order that compares attributes and models the routing decision procedure that compares routes using some combination of message fields. If attribute

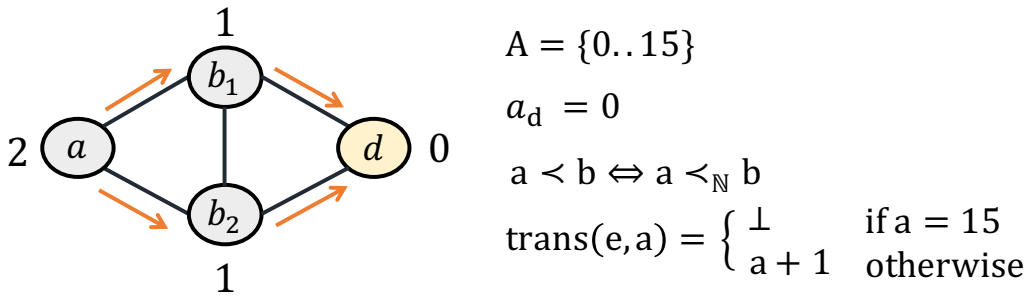


Figure 3.5: Modeling the RIP protocol as an SRP.

$a_1 \prec a_2$, then intuitively this means that a_1 is more desirable than a_2. We typically lift this comparison to work with the $\perp$ value by making it so that $\perp \prec a$ for all other attributes a. Finally, trans represents the transfer function that describes how attributes are modified (or dropped) when passed between routers. Given an edge and an attribute from the neighbor across the edge, it determines what new attribute is received at the current node. The transfer function depends on both the routing protocol and node's configuration.

Example: Figure 3.5 shows an example of an SRP for the RIP distance vector protocol. In RIP routes are chosen by hop count to the destination, but RIP uses a maximum hop count of 15, so we model the set of attributes as numbers between 0 and 15. The initial route has a path cost of 0, and the transfer function trans simply adds one to the attribute along an edge so long as it is no greater than 15. Otherwise, the route gets dropped ($\perp$). Finally, the comparison relation $\prec$ uses the standard comparison on natural numbers.

3.4.2 SRP Solution

Given an SRP instance, we can describe its (possibly many) solutions. Intuitively, each solution is derived from a set of constraints that requires that each node be *locally stable*, *i.e.*, it has no incentive to deviate from its current best neighbor. For shortest path routing, an SRP solution will be a rooted shortest path tree. For policy-based routing such as BGP, the paths may not be the shortest paths but will still form what is called a "stable tree" [56].

Definition: Formally, an SRP solution is an attribute labeling $\mathcal{L} : V \to A$ that maps each node to a final route (attribute) chosen to forward traffic. A solution then is any labeling $\mathcal{L}$ that satisfies the constraints shown in Figure 3.4. Namely, the labeling of the destination node d should be the special attribute a_{d}. If there are no attributes available from neighbors ($\mathsf{attrs}_{\mathcal{L}}(u) = \emptyset$), then node u has no route to the destination ($\perp$). Otherwise, $\mathcal{L}(u)$ is chosen to be an attribute choice that is minimal according to the comparison relation ($\prec$). If there is more than one minimal attribute, then any minimal value can be chosen. The set of attributes at a node stems from the choices set from neighbors. The choices is defined as follows: for each edge $e = (u, v)$ from u, apply the transfer function from the neighbor's label to obtain a new attribute $a = \mathsf{trans}(e, \mathcal{L}(v))$, ignoring any attributes that get dropped ($a = \perp$).

Given an SRP solution, it is easy to reconstruct the forwarding behavior. We can define $\mathsf{fwd}_{\mathcal{L}}(u)$ as the set of edges e such that u forwards traffic to the destination over e. $\mathsf{fwd}_{\mathcal{L}}(u)$ is defined such that the attribute learned from e is equal to the best choice $\mathcal{L}(u)$ at u. If there is more than one such choice, then a node may forward to multiple neighbors.

Example: In Figure 3.5, there is a single (in this case unique) solution $\mathcal{L}$, where $\mathcal{L}(v)$ that is shown annotated next to each node v. The destination node d has hop count 0, both nodes b_i have hop count 1, and node a has hop count 2. The induced forwarding relation is shown with the arrows. Router a will forward to both neighbors b_i using multipath routing, and each b_i will forward to the destination d. If we wanted to not allow multipath routing in the SRP, we would need to include the router ID as part of the comparison relation ($\prec$).

3.4.3 Modeling Common Routing Protocols

The SRP formulation can faithfully model many common routing protocols. For simplicity, we assume for now that the network runs only one routing protocol; we will consider multi-protocol networks and other configuration primitives in Section 3.6.

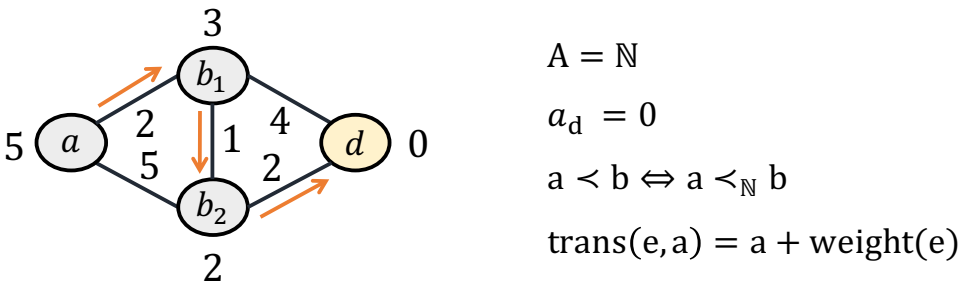


Figure 3.6: Modeling the OSPF protocol as an SRP.

Distance vector: There are several *distance-vector* protocols like EIGRP and RIP that compute variations of shortest paths to the destination. These protocols can be modeled similarly to the example with RIP where a cost is updated along each edge in the transfer function and the comparison relation $\prec$ is based on a numeric comparison.

OSPF (link state): Open Shortest Path First is a popular link state protocol where routers exchange link cost and status information. Although the implementation mechanism is vastly different from a distance-vector protocol, we can model OSPF in a similar way. Figure 3.6 shows an example of an SRP for OSPF. The attribute set $A = \mathbb{N}$ is any natural number and represents paths cost; the comparison relation compares this cost; and the transfer function adds the (configured) link cost. A large OSPF network may be split into multiple areas where each device prefers intra-area routes over inter-area ones. We can model this behavior using attributes that are tuples of the path cost and a boolean that indicates whether it is an inter-area route. The comparison relation prioritizes intra-area routes followed by path cost, and the transfer function changes the boolean value when crossing an inter-area edge.

BGP (path vector): For BGP, we assume (for now) that all routers use their own AS number, *i.e.*, eBGP (as in large data centers [71]) and discuss iBGP in Section 3.6. We model eBGP using $A = \mathbb{N} \times 2^{\mathbb{N}} \times list(V)$, where the components are: (1) a local preference value, (2) a collection of community tags, and (3) a list of nodes defining the AS path. Other BGP attributes such as MEDs or origin type can be similarly modeled, but are omitted for simplicity. BGP's comparison function

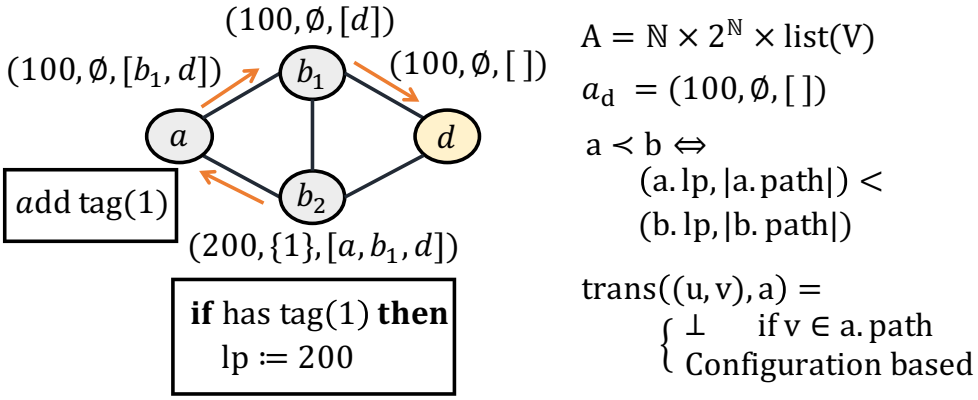


Figure 3.7: Modeling the BGP protocol as an SRP.

first compares local-preference followed by the AS path length. Its transfer function appends the current AS to the AS path when exporting a route. It also drops attributes that form a loop when the current node is present in the AS path. Otherwise, the router's policy, per its configuration, is applied.

Figure 3.7 shows an example, where $a.\mathrm{lp}$ and $a.\mathrm{path}$ denote components of an attribute $a = (\mathrm{lp}, \mathrm{tags}, \mathrm{path})$. Assume that in this network b_2 prefers going through a to reach destination d and that this policy is achieved by configuring a to add tag 1 to outgoing messages and configuring b_2 to prefer this tag. The configuration-driven part of the transfer function is shown in the boxes for routers a and b_2. Router a adds the tag 1 to attributes it exports; and b_2 checks for this tag, and if present, assigns a higher (better) local preference value than the default value (100), which ensures that b_2 prefers to go through a. The arrows in the figure indicate the final forwarding behavior of this network, and a solution labeling $\mathcal{L}$ is shown next to each node.

Static routing: Operators configure static routes that describe which interface to use for a given destination. Figure 3.8 shows an example where routers a and b_2 are configured with static routes. We model static routing using the set of attributes $A = \{true\}$ which indicates the presence of a static route. Since there is only one attribute, the comparison relation is trivially empty. The

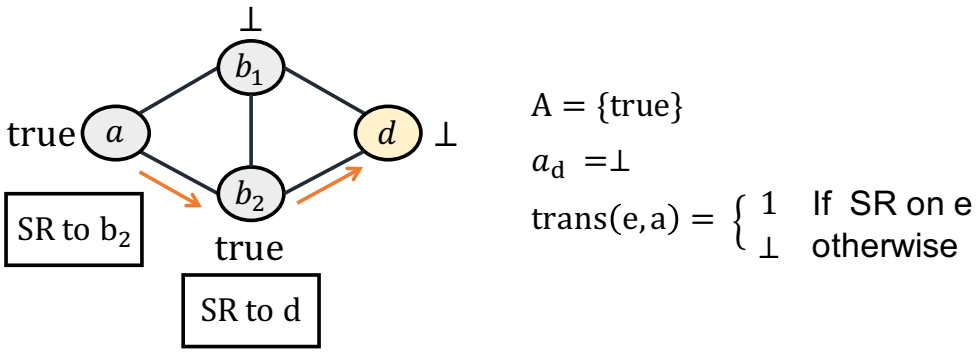


Figure 3.8: Modeling Static Routing as an SRP.

transfer function does not depend on the neighbor at all; it returns $true$ if there is a static route configured locally along an edge and $\perp$ otherwise.

3.5 Translation to SMT

The Stable Routing Problem provides a foundation that can be used to translate a network into logic. Since the solution to an SRP is defined as a set of logic constraints, we can encode an SRP using off-the-shelf SMT solvers to be able to automatically reason about network properties for all possible data planes provided that the $\prec$ relation and the trans function can be encoded in logic. In this section we describe the implementation details necessary to translate a real network to SMT using SRP as a foundation.

3.5.1 Overview

While the SRP gives a logical account for the different stable forwarding behaviors in the network, it does not tell us how we can verify properties such as "router R1 can always reach host2". To do this, we generate F, a system of SMT constraints defined as the conjunction of the formula N, the behavior of the network as an SRP from the current configurations of all routers, and $\neg P$, a negated property of interest to the operator (*e.g.*, reachability). Once encoded in logic an SMT

solver will search for a satisfying solution to $F = N \wedge \neg P$. Because N and P will be encoded in decidable fragments of logic, there are only two possible outcomes:

1. The solver finds a satisfying solution to F, which means there is a stable data plane that can arise (according to N), where P does not hold.

2. The solver proves that F is unsatisfiable, meaning there is no stable data plane that can arise where P does not hold.

As we will see, because N will take into account things like external peer messages, we can get a guarantee that P holds for all possible environments. A limitation of this type of encoding in SMT is that we can not actually check if the network will converge to a stable solution – only that P will hold if it converges. For example, there are networks that are known to never stabilize [56], and so we would trivially find that P holds in every stable state for such networks since there are no stable states.

3.5.2 Design Decisions and Limitations

Our verification approach is general and flexible, but it does have several limitations. The most critical design choice involves the fact that our system describes the stable solutions to which the control plane will converge; it does not simulate the execution of the control plane as a message-passing system. This choice improves performance, but it also means we give up the possibility of verifying properties about transient states of the network prior to convergence. Many other verification tools such as ERA [40] and ARC [49] share this limitation.

A second important design decision is that we only consider elements of the control plane that influence the forwarding decisions pertaining to a *single* symbolic packet at a time. As a result, it ends up being more expensive to model a few features that introduce dependencies among destinations. For example, it is possible for static routes to specify a next hop IP address that does not belong to a directly-connected interface, thereby requiring the model to understand how to route to that next hop. In this case, we will see that we must create a separate copy of every

control plane encoding variable to determine the forwarding for a second packet corresponding to the next hop address. Likewise, modeling iBGP requires one additional copy of every control plane variable for every router configured with iBGP. This additional complexity appears inherent since such features introduce cross-destination dependencies. We are not aware of any other verification tool that models them at all. Nevertheless, the cost of doing so will decrease the scalability of our system when these features are used.

3.5.3 Encoding the Packet

One challenge in translating a network to SMT is that there are many different destination IPs being routed simultaneously. Our formulation of an SRP on the other hand assumed that the destination was fixed ahead of time and its location was known. However enumerating all such destinations would be highly intractable. Instead, we opt to treat the destination itself as being symbolic. In particular, we start by modeling a symbolic packet (including its destination IP, source IP, etc.) and then encode the control plane behavior based on what IP ranges the packet's destination IP falls into.

The first section of Figure 3.9 lists the variables used to represent a symbolic packet. The packet's destination IP is modeled by an integer variable dstIP, which ranges from 0 to $2^{32}-1$. We model other fields similarly. If operators wish to ask a question about a specific destination, such as 10.0.0.0, they may issue a query that constrains our model to consider only packets with that destination (*e.g.*, using the formula dstIP $=$ 10.0.0.0 in their property P). If they instead wish to ask about packets with *any* destination IP, they may leave the dstIP field unconstrained. Traditional (non-SDN) networks do not typically modify packet headers (except for TTL and CRC fields, which we do not currently model)—they only forward or block them. Consequently, we use only one, global copy of each of these packet variables in our encoding.

Variable	Description	Representation
Data plane		
dstIp	Packet destination IP addr	$[0, 2^{32})$
srcIp	Packet source IP addr	$[0, 2^{32})$
$\mathrm{dstPort}$	Packet destination port	$[0, 2^{16})$
$\mathrm{srcPort}$	Packet source port	$[0, 2^{16})$
$\mathrm{protocol}$	Packet Protocol	$[0, 2^{8})$
$\mathrm{icmpType}$	Packet ICMP type	$[0, 2^{8})$
$\mathrm{icmpCode}$	Packet ICMP status code	$[0, 2^{4})$
$\mathrm{tcpFlags}$	Packet TCP fields	8 bits
Control plane		
valid_a	True if a is bot $\perp$	1 bit
prefix_a	Prefix for attribute a	$[0, 2^{32})$
length_a	Prefix length for a	$[0, 2^{5})$
metric_a	Protocol metric for a	$[0, 2^{16})$
ad_a	Administrative distance for a	$[0, 2^{8})$
lp_a	BGP local preference for a	$[0, 2^{32})$
med_r	BGP MED attribute for a	$[0, 2^{32})$
$\mathrm{bgpInternal}_a$	Was r learned via iBGP	1 bit
$\mathrm{igpMetric}_a$	IGP metric cost for iBGP	$[0, 2^{16})$
$\mathrm{originatorId}_a$	Route reflector originator ID	$[0, 2^{16})$
$\mathrm{community}_{a,c}$	Is community c attached to a	1 bit
$\mathrm{ospfType}_a$	OSPF area type	$[0, 2^{2})$
rid_a	Neighbor router ID for a	$[0, 2^{32})$
Decision		
$\mathrm{controlfwd}_{x,y}$	x fwds to y (ignores ACLs)	1 bit
$\mathrm{datafwd}_{x,y}$	x fwds to y (includes ACLs)	1 bit
Topology		
$\mathrm{failed}_{x,y}$	Is the link from x to y failed	$[0, 1]$

Figure 3.9: Selected symbolic variables from the model

3.5.4 Encoding the Control Plane

We now have a representation for a data packet, but in order to determine what happens to this (symbolic) packet in the network, we must of course model the various control plane protocols to understand how they decide to forward packets.

Multiple protocols: If the network was running only a single protocol for every device, then the SRP solution already describes exactly the constraints we need to encode the control plane

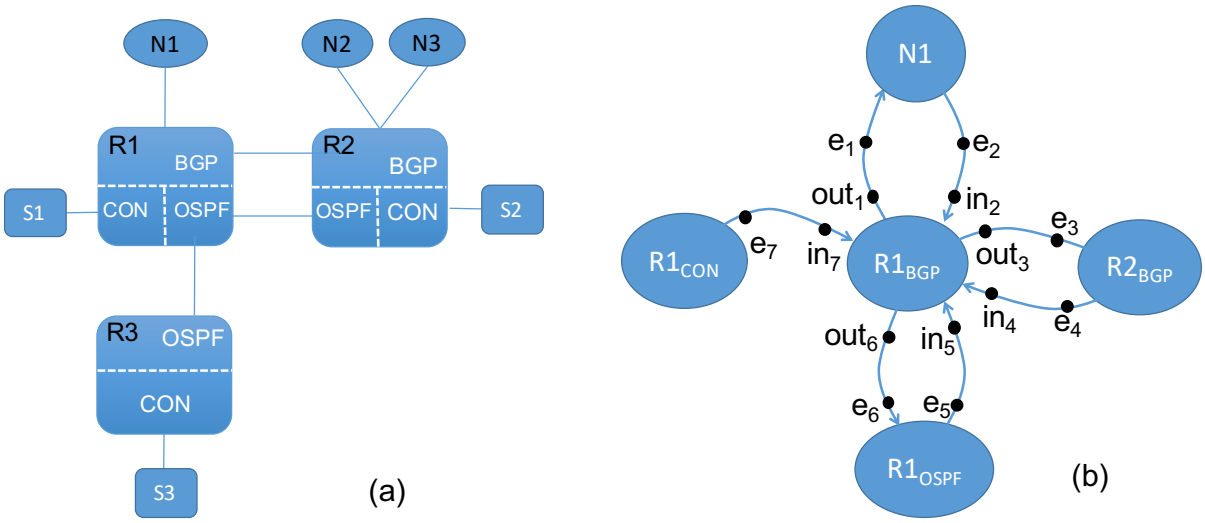


Figure 3.10: From the network in Figure 3.3, (a) Its protocol-level decomposition. (b) Routing information flow for BGP at R1.

dynamics. However, routers commonly run multiple protocol instances, each of which operates independently and selects a best route for a destination based on the information from its remote peers and redistribution from other local routing protocols.

To model these interactions, we treat every protocol *instance* as though it were its own device in the SRP. This is similar to network model used in ARC [49], which models each protocol instance as its own router. As an example, consider the network in Figure 3.10(a), which is a protocol-level view of the original example from Figure 3.3. In the protocol-level view of the network, we can see each protocol instance running on each router, and how they are connected to other protocol instances. For example, the BGP instance on routers R1 and R2 are connected and will exchange routes with each other, as well as with external neighbors N1-3. R1 and R2 also are both running OSPF instances that also communicate with each other. CON denotes directly connected routes, *i.e.*, those routes known from a directly connected interface. We model them as if they are another protocol to avoid special cases.

Figure 3.10(b) then splits each protocol instance into its own node to create a protocol-level SRP instance where the nodes are protocol instances. For simplicity, the graph is shown zoomed in on R1's BGP instance. Each edge in this new graph represents information flow between different

protocol instances. For example, the nodes R1_{OSPF} and R1_{BGP} represent protocols OSPF and BGP on router R1. Since OSPF redistributes into BGP, and vice versa, there are edges back and forth between R1_{OSPF} and R1_{BGP}. The outgoing edge from R1_{CON} indicates that the connected routes are redistributed into BGP. Since R1 uses BGP with the external neighbor N1 and R2, there are edges in both directions between R1_{BGP} and N1 and R2_{BGP}.

Encoding message attributes: In the SRP definition, we use the set A to represent the format of protocol messages. To encode these attributes in SMT in practice, we use a collection of symbolic values taken from integer and boolean domains. As with the data packets, constraints may map these variables to specific concrete values (*e.g.*, the prefix 10.1.0.0/24) or may leave them fully or partially unconstrained.

The second section of Figure 3.9 lists the main fields we use for symbolic attributes. First, every attribute contains a special boolean field, called valid. If valid is true, then a message is present and the remaining contents of the attribute are meaningful; otherwise, they are not meaningful (*i.e.*, this is the lack of a route: $\perp$). Each attribute is for a destination prefix of a particular length. The metric is a protocol-specific measure of the quality of the route. For instance, it is path length for BGP and path cost for OSPF. Announcements for that prefix are also annotated with the administrative distance (ad). When multiple protocol instances offer a route to the same prefix, this measure determines which one is used for forwarding. These attributes also contain many protocol-specific fields such as the local preference (lp) for BGP, the BGP multi-exit discriminator (med), whether a BGP route was learned via iBGP (bgpInternal), the OSPF area type (ospfType) for OSPF, and so on. If the network is running multiple protocols, then the fields for each will be included in an attribute. The router id (rid) is used to break ties among equally-good routes.

Because we are interested in those attributes that are relevant for the (symbolic) destination for which the current control plane behavior is being determined by the solver, we need to connect the destination prefix in the SRP solution to the destination in the packet. To do this, we ensure that the valid field of a control plane attribute will be true if and only if i) a message is present (*e.g.*,

advertised from a neighbor and not filtered), and ii) the *control plane* destination prefix applies to the *data plane* destination IP of the packet of interest. We capture the latter dependence with the following logic constraint:

$$e.\text{valid} \implies \textbf{FBM}(e.\text{prefix}, \text{dstIP}, e.\text{length})$$

The function FBM (first bits match) tests for equality of the first e.length bits of the prefix (e.prefix) and destination IP, thus capturing the semantics of prefix-based forwarding. The constraint FBM(p_1,p_2,n) is actually surprisingly tricky to encode efficiently. A naive solution that represents p_1 and p_2 as bit-vectors of size 32 is slow. We describe a more efficient encoding in Section 3.8.

Encoding the transfer function: In the definition of the SRP, we treated the transfer function (trans) as a black box. However, for a practical implementation we need to be able to encode this transfer function in SMT. Recall that the transfer function takes in an attribute (A) as an input, along with a topology edge, and returns a new attribute $A_\perp$ that may be modified or dropped. However, in realistic router implementations, the transfer function along an edge $e = (u, v)$ is broken up into two parts: (1) an export filter from u, and (2) an import filter at v. To model this, we break up each transfer function into two parts that represent an attribute being partially evaluated along an edge. The edge labels in Figure 3.10(b) indicate the presence of such a partially evaluated attribute. Consider the edge between R2_{BGP} and R1_{BGP}. The label e_4 represents the message exported by R2's BGP process on the link to R1; and the label in_4 represents the message after traversing R1's BGP import filter on the link from R2. Naturally, the messages defined by in_4 and e_4 are closely related. We encode the relationship using SMT constraints generated from import filters in R1's configuration.

Routing messages from the environment are represented as attributes from an external neighbor. For example, the attribute e_2 is the export from neighbor N1. When left unconstrained, it represents the fact that N1 could send any message.

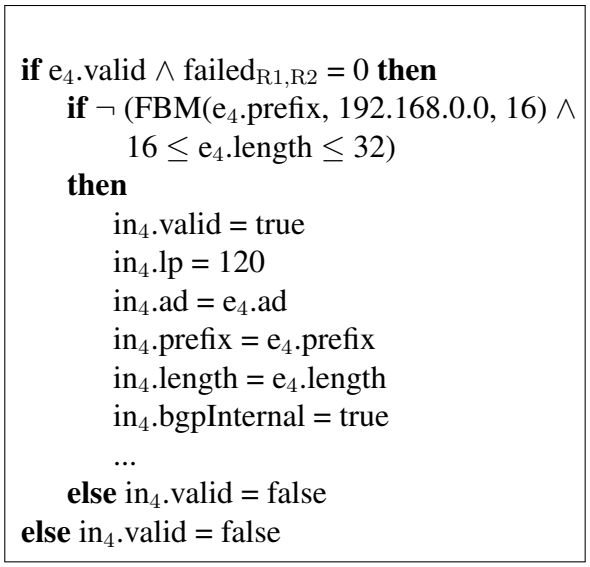


Figure 3.11: Translation of the R1 to R2 BGP import filter

Example: Import filter translation: Each router configuration defines (possibly per neighbor) filters that can either drop or modify protocol messages. As an example, consider the following configuration fragment for router R1.

```
ip prefix_list L deny 192.168.0.0/16 le 32
ip prefix_list L allow

route-map M 10
  match ip address prefix-list L
  set local-preference 120
```

This fragment blocks control plane announcements for any prefix that matches the first 16 bits of 192.168.0.0, and has prefix length between 16 and 32. It sets the local preference attribute to 120 for any other prefix. Assuming R1's BGP process is configured with this fragment as an import filter, we use it to constrain the relationship between the symbolic attributes e_4 and in_4 in Figure 3.10(c). More specifically, the filter is realized by the formula shown in Figure 3.11. The first line in this formula ensures that there can be an advertisement at in_4 only if R2 exports an advertisement to e_4 and the R1–R2 link is not failed. The second condition implements the import filter. If the two conditions are met, then information from R2 will arrive at R1. Hence, we set the valid bit of in_4, constrain the local preference to 120, and constrains in_4's other fields to be the same as e_4's. In all other cases, no advertisement arrives at R1, so its valid bit is set to false.

Such translation of import filters to symbolic constraints can also capture route redistribution between protocols. Users can set custom metric and administrative distance values for route redistribution, which would be updated as before.

Encoding the Comparison Relation ($\prec$): Each protocol instance selects a best route for each IP prefix among those available. For example, the routes available to $\mathrm{R1}_{\mathrm{BGP}}$ include routes from its neighbors and thus defined by the status of the symbolic attributes in_2, in_4, in_5, and in_7. The available routes are ordered by the decision process in a standard way. For instance, BGP first prefers the route with the highest administrative distance, and if those are equal, the highest local preference, then highest metric, *etc*. The $\prec$ relation for BGP is therefore a lexicographic ordering of several attribute fields. We use a recursive encoding in SMT as follows:

$$a_1 \prec a_2 \equiv \big(a_1.ad < a_2.ad \vee (a_1.ad = a_2.ad \wedge (a_1.lp > a_2.lp \vee \ldots))\big)$$

Here an attribute is preferred in BGP if it has a lower administrative distance (ad), or if they are equal and it has a higher local preference, or their local preferences are equal and so on, where the pattern repeats for each field used in BGP to compare messages. When implementing $\prec$ for a protocol, we only compare fields relevant for that protocol. For instance, the ospfType field would not be compared for BGP nodes.

Encoding the labelling ($\mathcal{L}$): To encode the best route as defined by $\mathcal{L}$, we need to be able to choose the minimum value among neighbors according to $\prec$. The selected route will be the one that is both available (the valid bit is set) and is best according to $\prec$. Logically, our encoding introduces a new symbolic attribute $\mathrm{best}_{\mathrm{prot}}$ for each protocol instance prot represented by node v, which is used to record the value of $\mathcal{L}(v)$.

A naive encoding of "best" that does a pairwise comparison of all messages from neighbors has quadratic size and would result in very poor performance. However, assuming that at least one neighbor has a route to the destination, we can encode the "minimum" function used in the

definition of $\mathcal{L}$ in logic with a number of constraints linear in the number of neighbors:

$$\bigwedge_{i \in \{2,4,5,7\}} \text{best}_{\text{BGP}} \preceq \text{in}_\text{i} \quad \wedge \quad \bigvee_{i \in \{2,4,5,7\}} \text{best}_{\text{BGP}} = \text{in}_\text{i}$$

This constraint encodes the minimum attribute from neighbors at a node by stating that it is less than or equal to all alternatives and equal to at least one of them. The less than operation in the minimum is encoded using the comparison relation ($\prec$) shown previously. However, this does not take into account the possibility that there might not be a route from any neighbor (*i.e.*, the choices set from the definition of $\mathcal{L}$ might be empty).

We can generalize the encoding to account for this possibility by taking into account by computing the minimum only among those routes that are not $\perp$ as follows:

$$(\bigvee_i \text{in}_\text{i}.valid) \iff \text{best}_{\text{prot}}.valid$$

$$\text{best}_{\text{prot}}.valid \implies \bigwedge_i (\text{in}_\text{i}.valid \implies \text{best}_{\text{prot}} \preceq \text{in}_\text{i})$$

$$\text{best}_{\text{prot}}.valid \implies (\bigvee_i \text{in}_\text{i}.valid \vee \text{best}_{\text{prot}} = \text{in}_\text{i})$$

If there is some valid advertisement, then this router will have a valid route, which will be the minimum among valid routes. Otherwise, this router will not have a valid route.

Encoding the forwarding: Each router installs only one route in its data plane, which is then used to forward traffic. Thus, it chooses a best route among all routing protocols. Once again, this can be modeled with a new symbolic record $\text{best}_{\text{overall}}$, which is similarly constrained to be the best among all the $\text{best}_{\text{prot}}$ attributes. To represent the final forwarding decision of the router, we introduce a new boolean variable $\text{controlfwd}_{\text{x,y}}$ for each edge in the network between routers x and y. The variable indicates that router x decides to forward traffic for the destination to router y. Recall that forwarding goes in the opposite direction of control plane messages. Intuitively, router x will decide to forward to router y if the message received from y is equal to the x's best choice.

```
if best_BGP.valid ∧ failed_R1,R2 = 0 then
    if ¬ best_BGP.bgpInternal ∧ best_BGP.length + 1 ≤ 255
    then
        out_3.valid = true
        out_3.lp = best_BGP.lp
        out_3.ad = best_BGP.ad
        out_3.prefix = best_BGP.prefix
        out_3.length = best_BGP.length + 1
        ...
    else out_3.valid = false
else out_3.valid = false
```

Figure 3.12: Translation of the R1 to R2 BGP export filter

For example, to determine if R1 will forward to R2, we use the following constraint:

$$\text{controlfwd}_{\text{R1,R2}} = (\text{e}_4.valid \wedge \text{e}_4 = \text{best}_{\text{overall}})$$

Example: Export filter translation: Now that we have encoded the labelling $\mathcal{L}$ symbolically, we can finish encoding the export portion of the transfer function. After selecting a best route, each protocol will export messages to each of its peers after potentially processing these messages through peer-specific export filters. Figure 3.12 shows the route export constraint for R1_{BGP}'s export to R2_{BGP} assuming the default export filter. The encoding of route export is similar to that of an import filter, but with some differences. First, the export constraint will connect the attribute for the protocol's best route (best_{BGP}) with an attribute on an outgoing edge of a router (e.g., out_3). Second, the route export constraint accounts for the fact that iBGP routes should not be re-exported to other iBGP peers by checking if the best route was learned via iBGP. Third, the path metric is updated according to the protocol (e.g., adding 1 for BGP). Finally, the route is only exported if the new path metric does not overflow the maximum protocol path length (e.g., 255 for BGP).

3.5.5 Encoding the Data Plane

Although routers decide how to forward packets in the control plane through their decision process, the actual data plane forwarding behavior can differ due to the presence of an access control list (ACL), which lets a router block traffic directly in the data plane. To handle ACLs, we create additional variables to represent the final data plane forwarding behavior of the network. For each variable $\text{controlfwd}_{x,y}$, we create a corresponding $\text{datafwd}_{x,y}$ variable. The data plane forwarding will be the same as the control plane forwarding modulo any ACLs. For example, consider the following ACL:

```
access-list 1 deny ip  172.10.1.0  0.0.0.255
```

The mask `0.0.0.255` signifies the wildcard bits for the match. This ACL will thus block any packets that match destination IP `172.10.1.*` in the data plane. This constraint is captured by first translating the ACL to a formula and then conjoining it with the control plane decision in the following way:

$$\text{datafwd}_{R1,R2} = \text{controlfwd}_{R1,R2} \wedge \neg \text{FBM}(\text{dstIP}, 172.10.1.0, 24)$$

3.5.6 Encoding Properties

The model above captures the joint impact of all possible network interactions, but to verify properties of interest we can instrument it with additional variables as needed. For example, suppose we wish to check that router R3 can reach N1 regardless of any advertisements received from neighbors N2 and N3. For each router x in the network, we add a variable reach_x representing that x can reach the destination subnet. For R1, which is directly connected to N1, we add:

$$\text{canReach}_{R1} \iff \text{datafwd}_{R1,N1}$$

For every other router, we say it can reach N1 if it can forward to some neighbor that can reach N1. For router R3:

$$\text{canReach}_{\text{R3}} \iff \bigvee_{R \in \{R1\}} (\text{datafwd}_{\text{R3,R}} \wedge \text{canReach}_{\text{R}})$$

Since we are interested in checking that the property holds for any possible packet, we leave the packet fields (e.g., dstIp) unconstrained. Finally, we would assert the negation of the property we are interested in, namely $\neg\text{canReach}_{\text{R3}}$ and ask the solver to prove unsatisfiability, thereby ensuring that the property holds for all packets and environments.

3.6 Generalizing the Model

While Section 3.5 gave an overview of the basic encoding of a network into SMT, it elided many details about how to handle some of the most commonly used configuration features. This section describes how we can generalize the previous encoding to model a number of additional features and protocols.

3.6.1 Route redistribution

Although a router might participate in routing with multiple protocols, it can only install a single forwarding entry in its forwarding table. In order to ensure that there is always a "best" route, vendors use *administrative distance*, which measures the relative trustworthiness of particular routes. Each protocol has a default administrative distance, which can be overwritten in the configuration by the operator. The route with the lowest administrative distance will be used. We model modifications to the administrative distance in the transfer function between SRP nodes that represent different protocols.

3.6.2 Static route recursive lookup

Although we model static routes as an SRP where the next-hop node is known, most vendors provide a mechanism to allow static routes to lookup the next hop based on the routing for another destination IP address. To resolve the forwarding behavior, the router will then recursively lookup the next hop for this address. However, in our model, because we are only determining the forwarding behavior for one destination at a time, we can not directly determine the next hop. To handle this case, we must make two copies of the network encoding: One for the current destination, and another to determine the forwarding for the static route's next hop address.

3.6.3 Aggregation

Aggregation, in which routers announce a less-specific prefix that covers many, more-specific prefixes, helps reduce the size of the routing tables. When misconfigured, routers that should aggregate prefixes may instead advertise a large number of more-specific prefixes. Such misconfigurations are insidious because they can lead to uneven traffic distribution (when aggregation is not applied consistently) and high CPU utilization on the router due to the large influx of route advertisements for more specific prefixes. We model aggregation as a modification to the prefix length attribute. If a prefix is valid for the destination IP address before aggregation, it remains valid after aggregation, but with a shorter prefix length. For example, if a /24 prefix is relevant for the packet's destination IP then so is its aggregated /16 prefix.

3.6.4 Multipath routing

The encoding in Section 3.2 assumed that routers select a single best path, but multipath routing, where traffic is spread over multiple equally-good routes to balance load, is common in modern networks. To encode multipath routing, we relax the best route comparison so that it does not compare the router ID. This relaxation no longer requires a total ordering of preferred routes, and any route as good as the best route will be used.

3.6.5 BGP community regexes

BGP communities are strings that can be attached to (or removed from) route advertisements. For many vendors, it is possible to check if there exists a BGP community attached to a message that matches a regular expression using what are called "extended" communities. To encode extended communities, we statically evaluate each extended community regex that appears in a configuration against all other communities defined in the configurations. We then replace a community regex such as: `.* 3 .*` with a test using a disjunction of individual communities, *e.g.*, checking if any community in the set $\{13, 23, 331\}$ is attached.

3.6.6 iBGP

Modeling iBGP is challenging because it introduces cross-destination dependencies through recursive lookup. In order to determine the forwarding behavior for a particular packet p over a network using iBGP, one first has to determine the forwarding behavior for each user-defined next-hop destination IP address configured between iBGP peers. For example, if router A has no IGP route to router B's iBGP-configured next-hop IP address, then the peers can not exchange BGP advertisements about packet p.

To model iBGP, we create N additional copies of the network where N is the number of routers configured to run iBGP. Each copy of the network encodes the forwarding behavior for a packet destined to the next-hop IP address associated with one of the iBGP-configured routers. We add the constraint that router A only propagates routes to router B over an iBGP connection if A can reach B in the network copy corresponding to B's configured next-hop destination IP address.

The variable $\mathrm{bgpInternal}$ indicates whether or not a route was learned from an iBGP peer. Routes learned via iBGP are allowed to be exported to eBGP peers but not to other iBGP peers. If a router decides to forward traffic to an iBGP peer, we lookup the actual IGP forwarding behavior from the copy of the network corresponding to that neighbor's next hop destination IP address.

3.6.7 Route reflectors

Route reflectors help scalably disseminate iBGP information among BGP routers by acting as an intermediary. To model route reflectors, we use a slightly modified scheme from that described above for iBGP. Each symbolic attribute includes a variable ($\mathrm{originatorId}$) indicating the router that initially sent the advertisement. Routes are then exported according the route-reflector semantics (*e.g.*, route reflectors reflect routes with a Non-Client $\mathrm{originatorId}$ to Clients). Client routers then lookup next-hop forwarding reachability based on the copy of the network corresponding to the value of $\mathrm{originatorId}$. Loops (*e.g.*, those prevented with the CLUSTER_ID field) are handled similarly to BGP (see Section 3.8).

3.6.8 Multi-exit discriminator (MED)

The MED attribute of BGP routes allows an AS to indicate preferences for paths for incoming traffic (*i.e.*, "cold potato" routing). There are multiple ways in which MEDs may be used by a router depending on the configuration options and router vendor. In one usage, the MED values are compared independent of the next-hop AS. We model this case by ensuring that MEDs are compared when computing the best route (*e.g.*, $\mathrm{best}_{\mathrm{BGP}}.\mathrm{med} \leq \mathrm{in}_1.\mathrm{med}$). In another usage, the MED values are compared only for routes with the same next hop AS. To model this case, we first add a variable to each symbolic control plane attribute that "remembers" what neighboring AS the route was learned from. The import function from an external neighbor will set the value of the next hop AS. The best route constraints then only compare the MED when the AS is the same. For example, we generate the constraint:

$$(\mathrm{best}_{\mathrm{BGP}}.\mathrm{asn} \neq \mathrm{in}_1.\mathrm{asn}) \vee (\mathrm{best}_{\mathrm{BGP}}.\mathrm{med} \leq \mathrm{in}_1.\mathrm{med})$$

In yet another usage, the age of a route determines the route comparison order, which means that routes with worse MED values may be chosen over those with better values even when the routes

have the same next hop AS. Rather than model the age of each route, we overapproximate this behavior by selecting any best route without comparing MEDs.

MEDs are also non-transitive, i.e., the AS that receives them does not export them to other ASes. We model non-transitivity similarly to iBGP. We add a variable indicating whether a MED was learned from an external peer, or set within the current AS. Routes with MEDs learned from a peer are not exported to other ASes.

3.7 Property Expressiveness

The SMT encoding of the control plane presented in Section 3.5 is highly flexible. It is easy to add new constraints and compose them with the symbolic control plane model using the standard logical conjunction and disjunction operators. We can leverage this flexibility to verify a wide variety of properties for real networks.

3.7.1 Reachability and isolation

We focus on answering reachability queries for a fixed destination port and set of source routers. To answer such a query, each router x is instrumented with an additional variable $\mathrm{canReach}_x$ representing the fact that the router can reach the destination port. We then add constraints similar to the example from Section 3.2. Isolation is checked by simply asserting that a collection of routers are not reachable.

One benefit of the graph-based encoding is that queries can involve many routers at once and the solver will analyze their joint impact. For example, to check if two routers r_1 and r_2 can both either reach or not reach the destination, one would assert $\mathrm{canReach}_{r_1} \iff \mathrm{canReach}_{r_2}$. Similarly, the user can check if all routers from a set S can reach the destination in a single query by checking: $\bigwedge_{s \in S} \mathrm{canReach}_s$. In contrast, in existing data plane and control plane verification tools, to answer questions about reachability between all pairs of n devices, one is often required to run n^2 separate queries, which can be very expensive [84].

3.7.2 Waypointing

Suppose we want to verify that traffic will traverse a chain of devices $m_1, \ldots, m_k$. Rather than adding one variable for each router as with reachability, instead we add k variables for each router to indicate how much of the service chain has been matched. If a router forwards to neighbor m_j and its $(j-1)$th variable is true, then the jth variable must be true for that router. Routers where the kth variable is true will send traffic through the service chain.

3.7.3 Bounded or equal path length

In many settings, it is desirable to guarantee that traffic follows paths of certain length. For example, for a data center with a folded-CLOS topology, an operator may wish to ensure that traffic never traverses a path longer than four hops. A violation of such an invariant likely indicates a configuration bug. Similarly, the operator may want to ensure that all top-of-rack routers in a pod use equal length paths to the destination. Similar to reachability, path length is easily instrumented in the model by adding a new integer variable for each router in the network. Each router has path length n to the destination if it forwards to some neighbor with path length $n-1$.

3.7.4 Disjoint paths

It is possible to ensure that two different routers use edge-disjoint paths to a destination. Given two routers, we add two bits to each edge indicating whether either router ever forwards through that edge. A constraint then states that both bits are never set for any edge. A similar approach can be used to guarantee that paths do not share nodes or other shared-risk elements (*e.g.*, fiber conduits), by introducing a variable for each risk factor.

3.7.5 Forwarding loops

Forwarding loops in the network can arise from configuration errors when using features like route redistribution and static routes. To detect forwarding loops for a particular router r, we add a

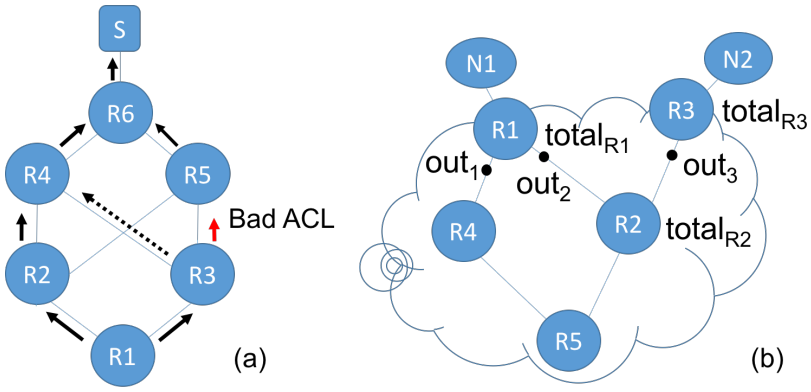


Figure 3.13: Example networks for property encodings.

single control bit to say whether each other router will eventually send traffic through r. If r sends traffic to any neighbor with this bit true, then there will be a forwarding loop. As an optimization, we analyze configurations to identify routers where a forwarding loop is possible (*e.g.*, due to the presence of static routes). We then add control bits only for these routers.

3.7.6 Black holes

Black holes occur when traffic is dropped because it arrives at a router that does not have a corresponding forwarding entry. This behavior may be intentional (*e.g.*, in the case of ACLs) or unintentional. We can find black holes by checking if any router has a neighbor that forwards to it, yet the router itself does not forward to any neighbor.

3.7.7 Multipath consistency

Batfish [44] introduced a property called multipath consistency, which ensures that traffic along all paths from a source is treated the same. A violation of multipath consistency occurs when traffic is dropped along one path but not the other. Consider the example in Figure 3.13(a). Router R1 is configured to use multipath routing, yet an ACL on router R3 prevents traffic from using the link

to R5. We encode multipath consistency as follows.

$$\begin{aligned}
\text{canReach}_{R1} \implies & \bigwedge_{R \in \{R2,R3\}} \\
& (\text{controlfwd}_{R1,R} \implies \\
& \quad \text{datafwd}_{R1,R} \wedge \text{canReach}_R) \\
\text{canReach}_{R3} \implies & \ldots
\end{aligned}$$

The first constraint says that if R1 can reach the destination S at all, then forwarding to R2 (R3) in the control plane implies that R2 (R3) should also be able to reach the destination, and this also aligns with forwarding in the data plane to R2 (R3). In the example presented in Figure 3.13(a) this constraint will fail since R3 cannot reach the destination, due to the bad ACL to R5. Suppose now that R3 can also use multipath routing, and can therefore reach the destination via R4 (shown as the dotted edge). Now the first constraint at R1 will succeed, but the second constraint for R3 will fail, because R3 can forward through R4 but not through R5.

3.7.8 Neighbor or path preferences

Operators often want to enforce preferences among external neighbors based on commercial relationships. For example it is common to prefer routes learned from customers over peers over providers. Given a router R with three edges to neighbors N1, N2, and N3 with import attributes e1, e2, and e3, we can verify that N1 is preferred over N2 over N3 in the following way. For each neighbor, we add a constraint that, if a message survives the import filter, and all other more preferred neighbor advertisements do not, then the presence of the message implies that we will choose that neighbor in the selection process:

$$\begin{aligned}
\text{e1.valid} & \implies \text{controlfwd}_{R,N1} \\
\neg\text{e1.valid} \wedge \text{e2.valid} & \implies \text{controlfwd}_{R,N2} \\
\neg\text{e1.valid} \wedge \neg\text{e2.valid} \wedge \text{e3.valid} & \implies \text{controlfwd}_{R,N3}
\end{aligned}$$

This type of reasoning can be lifted to entire paths. For example suppose we want to verify that the network prefers to use $path_1 = x_1, \ldots, x_m$ over $path_2 = y_1, \ldots, y_n$. What we want to check is that if the less preferred path is used, then the more preferred path was not available:

$$\bigwedge_{i=1}^{n-1} \text{controlfwd}_{y_i,y_{i+1}} \implies \bigvee_{i=1}^{m-1} \neg \text{e}_i.\text{valid}$$

That is, whenever traffic flows along $path_2$ it is because $path_1$ is not available due the advertisement being rejected along one of the edges. A straightforward generalization of the above can help enforce preferences over classes of neighbors, instead of individual neighbors.

3.7.9 Load balancing

Consider the example network in Figure 3.13b. Suppose router R1 is configured to use ECMP to send traffic to R2 and R4. We can roughly model the effect of load distribution with the following steps. First, for each router R in the network we introduce a symbolic real number called total_R representing the portion of traffic going through R. For each source router of interest (*e.g.*, R1 and R3), we set the load to some initial value based on traffic measurements (*e.g.*, 1.0 in this example):

$$\text{total}_{R1} = 1.0 \wedge \text{total}_{R3} = 1.0$$

For each outgoing interface in the network, we add a variable out_i representing the fraction of the load sent out that interface, which depends on the forwarding behavior.

$$\begin{aligned} \text{out}_1 &= \text{if datafwd}_{R1,R4} \text{ then } x \text{ else } 0.0 \\ \text{out}_2 &= \text{if datafwd}_{R1,R2} \text{ then } x \text{ else } 0.0 \\ \text{total}_{R1} &= \text{out}_1 + \text{out}_2 \end{aligned}$$

Each interface's load is equal to the (same) value defined by a single new variable x if traffic is forwarded out the interface, otherwise it is 0. This new variable x ensures the loads are all equal

(this could be easily extended to weighted ECMP by scaling x by a constant according to the fraction of traffic split). The total at non-source routers is simply the sum of their incoming totals:

$$\mathrm{total}_{R2} = \mathrm{out}_2 + \mathrm{out}_3$$

Now we can ask questions about the load on each node/edge. For example, we can check that the difference between the loads on R2 and R4 is always within some threshold k:

$$-k \leq \mathrm{total}_{R2} - \mathrm{total}_{R4} \leq k$$

3.7.10 Aggregation and leaking prefixes

We can ensure that prefixes are aggregated properly (*e.g.*, a /32 is not leaked to an external network) by checking: whenever the network advertises attribute e to an external neighbor, then $\mathrm{e.length} = l$ where l is prefix length after aggregation.

3.7.11 Local equivalence

In many networks (*e.g.*, data centers), several devices will perform a similar "role" (*e.g.*, aggregation router) and have similar configurations. Checks for equivalence can help detect inconsistencies. For example, we might want to know that a particular community value is always attached to advertisements sent to external neighbors.

Because we fully model each router's interactions with all of its neighbors, we can check if two routers are behaviorally equivalent for some notion of equivalence. In particular, we ask if given equal environments (i.e., peer advertisements), the routers will make the same forwarding decisions and export the same new advertisements. For example, if two routers R1 and R2 both have the same two peers P1 and P2 with import attributes in_1 and in_2, and output attributes out_1

and out_2, then we check the following:

$$\begin{aligned} \mathrm{in}_1 = \mathrm{in}_2 \implies & (\mathrm{out}_1 = \mathrm{out}_2) \wedge \\ & (\mathrm{datafwd}_{R1,P1} = \mathrm{datafwd}_{R2,P1}) \wedge \\ & (\mathrm{datafwd}_{R1,P2} = \mathrm{datafwd}_{R2,P2}) \end{aligned}$$

3.7.12 Full equivalence

It is also possible to check full equivalence between two sets of router configurations. This is done in a similar way as the local equivalence check, by first making two separate copies of the network encoding, and then relating the environments. As before, we check that all the final data plane forwarding decisions and all exports to neighboring networks must be the same as a result.

3.7.13 Stability and Uniqueness

Because the SMT constraints for the network encode only stable solutions, we can check if the network will converge to a stable solution for a fixed environment simply by checking if the encoding is satisfiable. If the network is unstable, then there will be no stable graph, and thus no satisfying solution. Similarly, we can check if the network will converge to a *unique* stable solution in the following way. First, we find a stable solution, which consists of an assignment to each variable x_i in the model. Then we check satisfiability again after adding a blocking constraint of the form:

$$\mathit{blocking} = \bigvee_i \neg x_i$$

This tells instructs the solver to find another solution, which is not exactly the same as the previous solution. If the solver comes back with another solution, then there is more than one stable solution.

3.7.14 Wedgies

BGP wedgies [55] can occur when, after one or more links fail, the protocol converges to a new, less desirable state. When the links are restored, the protocol does not change back to the old, more desirable state. Therefore, a wedgie can only occur when the less desirable state is a stable solution. Furthermore, wedgies need not only be limited to the BGP routing protocol. For example, similar types of stability problems can arise due to the complex semantics of route redistribution. Minesweeper can detect wedgies by enumerating stable solutions using blocking constraints, and then checking if certain solutions are less desirable than others (*e.g.*, due to path length).

3.7.15 Fault tolerance

Configurations that work correctly in the absence of failures may no longer work correctly after one or more links fail. For each property above, we can verify that it holds for up to k failures by adding the following constraint on the number of links that are failed:

$$\sum_{(x,y)\in \text{edges}} \text{failed}_{x,y} \leq k$$

Because link failures are part of the network model, the solver will learn facts about the impact of failures on the rest of the network control plane. This behavior means that properties involving failures can often be checked more efficiently than iterating over failure cases using a failure-free model (*i.e.*, verifying a property multiple times independently, once for each failure case).

3.7.16 Fault-invariance testing

We can use the same strategy as equivalence checking to instead check if the same property holds in a single network regardless of failures. For example, even if we do not know whether two routers should be able to reach one another (a possible problem when analyzing networks without specifications), we can check that the two routers are reachable if and only if they are reachable

after any single failure. Such a test can find instances where network behavior differs after failures. To check fault-invariance with respect to a property P, we create two copies of the network. For the first copy, we require that there are no failures. For the second copy, we allow there to be any k failures. We then check that P holds in the first copy of the network exactly when it holds in the second copy.

3.8 Optimizations

While conceptually simple, the naive encoding of the control plane described in Section 3.2 does not scale to large networks. We present two types of optimizations that dramatically improve the performance of the control-plane encoding.

3.8.1 Hoisting

Hosting lifts repeated computations outside their logical context and precomputes them once. Two main optimizations of this class that we use are:

Prefix elimination: Our naive encoding does not scale well in large part because of the constraints of the form FBM($p1$, $p2$, n), which checks that two symbolic variables have the first n bits in common. The natural way to represent $p1$ and $p2$ for this check is to use 32-bit bitvectors and check for equality using a bit mask. However, bitvectors are expensive and solvers typically convert them to SAT. In our model, this would introduce up to 128 new variables for every topology edge in the network (4 attributes per edge) thereby introducing an enormous number of additional variables.

To avoid this complexity, we observe that the prefix received from a neighbor does not actually need to be represented explicitly. In particular, because we know (symbolically) the destination IP address of the packet and the prefix length, there is a unique valid, corresponding prefix for the destination IP. For example, if the destination IP is `172.18.0.4` and the prefix length is /24, and

the route is valid for the destination, then the prefix must be `172.18.0.0/24` (alternatives such as `172.18.0.1/24` are treated identically).

However, we must still be able to check if a prefix is matched by a router's import or export filter. Somewhat unintuitively, we can safely replace any filter on the destination prefix with a test on the destination IP address directly, thereby avoiding the need to explicitly model prefixes. Consider the following prefix filter:

```
ip prefix_list L allow 192.168.0.0/16 ge 24 le 32
```

Its semantics is that it succeeds only if the first 16 bits of 192.168.0.0 match the prefix, and the prefix length is greater than or equal to 24 and less than or equal to 32. In general, for a prefix filter of the form `P/A ge B le C` to be well formed, vendors require that $A < B \leq C$. A simple translation of this for SMT attribute e is:

FBM(e.prefix, 192.168.0.0, 16) $\wedge$ (24 $\leq$ e.length $\leq$ 32)

Suppose now, we replace the test on the prefix contained in the control plane advertisement with a test directly on the destination IP address of a packet of interest:

FBM(dstIP, 192.168.0.0, 16) $\wedge$ (24 $\leq$ e.length $\leq$ 32)

There are two cases to consider. First, if e.length is not between 24 and 32, then both tests fail, so they are equivalent. Suppose instead, e.length is in this range. Recall that, because we are considering a slice of the network with respect to the destination IP address, for the advertisement corresponding to e to be valid, it must be the case that the prefix contains the destination IP. That is: FBM(e.prefix, dstIP, e.length). However, because we know the prefix length falls in the range between 24 and 32, it must be greater than 16. Since the first bits up to the prefix length are common between the destination IP and the prefix, the first 16 bits must also be the same. Therefore the above substitution is equivalent.

Further, because the test FBM is now purely in terms of constants in the configuration (not the symbolic prefix length variable), we can represent the destination variable as an integer and

implement the test using the efficient theory of integer difference logic (IDL). Thus, we would test that:

$$(192.168.0.0 \leq \text{dstIP} < 192.168.0.0 + 2^{32-16}) \ \wedge (16 \leq \text{e}_4.\text{length} \leq 32)$$

Loop detection: In protocols that support policy-based routing (e.g., BGP), path length alone does not suffice to prevent loops. For this reason, BGP tracks the ASNs (autonomous system numbers) of networks along the advertised path and routers reject paths with their own ASN. We can model this by maintaining, for each BGP router, a control bit saying whether or not the advertised path already went through that router. However, doing so can be expensive since the number of control bit variables for the entire network encoding grows with the square of the number of routers. Instead, we observe that any BGP router that uses only default local preferences (*i.e.*, only makes decisions based on path length) will never select a route where it is already part of the AS path. This is because the path containing the loop is strictly longer than the path without the loop. For example, if AS 1 uses shortest path routing only, then the AS path 1 2 1 3 can never arise in our model since AS 1 would prefer the path 1 3 instead. Similarly, BGP local preferences for external neighbors and for iBGP peers will not create loops. This optimization makes it possible to forgo modeling loops in most cases.

3.8.2 Network Slicing

Slicing removes bits from the encoding that are unnecessary for the final solution. We use the following slicing optimizations:

- Remove symbolic variables that never influence the decision process. For example, if BGP routers never set a local preference, then the local preference attribute will never affect the decision and can be removed.
- Keep a single copy of import and export variables for an edge when there is no import filter on the edge. The two variable sets will simply be copies of each other.

- Keep a single, merged copy of the export attribute for a protocol when there is no peer-specific export policy.
- Do not model directly connected routes for a router whose interface addresses can never overlap with the destination IP range of interest to the query. For example, when checking reachability to a destination interface, many interfaces on other routers will never influence the routing behavior.
- Merge the data plane and control plane forwarding variables along edges that do not have ACLs.
- Merge per-protocol and overall best attributes when there is only a single protocol running on a router.

Together, these optimizations are effective at removing a lot of redundant information that the SMT solver might otherwise have to discover for itself.

3.9 Implementation

Minesweeper uses Batfish [44] to parse vendor-specific configurations. It then translates Batfish's representation into a symbolic model. To check model (un)satisfiability, we use the Z3 SMT solver [35]. Our encoding exploits Z3's support for integer difference logic, and its preprocessor. Our implementation supports most of the features and properties described in the paper. We have validated its correctness empirically by comparing its output to that of the Batfish simulator on a large collection of networks. As Batfish does not currently support IPv6, Minesweeper does not either. Minesweeper is available as open source software [11].

3.10 Evaluation

We evaluate Minesweeper by using it to verify a selection of the properties described in Section 3.7 on both real and synthetic network configurations. In particular, we are interested in measuring (1) the ability of Minesweeper to find bugs in real configurations, which are otherwise hard to find; (2) its scalability for answering various queries on large networks; and (3) the impact of the optimizations described in Section 3.8 on performance. All experiments are run on an 8 core, 2.4 GHz Intel i7 processor running Mac OSX 10.12.

3.10.1 Finding Errors in Real Configurations

We demonstrate Minesweeper's ability to find bugs in real configurations by applying it on a collection of configurations for 152 real networks. We obtained these from a large cloud provider, and they represent different networks within their infrastructure. The networks range in size from 2 to 25 routers with 1–23K lines of configuration each. The networks use a combination of OSPF, eBGP, iBGP, static routes, ACLs, and route redistribution for layer-3 routing and are part of a data set described in detail in prior work [50]. These networks have been operational for years, and thus we expect that all easy-to-find bugs have already been ironed out. This data set was also analyzed by ARC [49].

Properties checked: Since we do not have the operator-intended specifications, we focus on four properties expected to hold in such networks:

- Management interface reachability: All nodes in the network should be able to reach each management interface, irrespective of the environment. Management interfaces are used to log into the devices, manage their firmware and configuration, and collect system logs. Uninterrupted access to it is important for the network's security and manageability.
- Local equivalence: Routers serving the same role (e.g., as "top-of-rack") should be similar in how they treat packets. We identify routers in the same role by leveraging the networks'

naming convention and check that all pairs of routers in the network in a given role are equivalent.

- No blackholes: When traffic is dropped due to ACLs, such dropping should always occur at the edge of the network.

- Fault-invariance: All pairs of routers in the network should be reachable from one another if and only if they are reachable after a single failure. A violation of this property would indicate that the network is highly vulnerable to failures.

Violations: We found 67 violations of management interface reachability. In each case, the violation occurs because of a "hijack", i.e., external neighbors sending particular announcements. For example, an external BGP advertisement for the same /32 interface prefix with path length ≤ 1 would result in a more preferred route for the destination that would ultimately divert traffic away from the correct interface.

The checks for local equivalence revealed 29 violations. Upon further investigation, we found that each violation was caused by one or more exceptions in ACLs where almost all routers in a given role would have identical ACLs except for a single router with an extra or a missing entry. Such differences are possibly caused by copy-and-paste mistakes.

The blackholes check found 24 violations. Most violations were not serious issues with routing, but instead revealed optimization opportunities. Traffic being dropped deep in the network could have been dropped near the source.

We found no violations of fault-invariance.

3.10.2 Verification Performance

We evaluate the performance of Minesweeper to verify different properties on real and synthetic configurations.

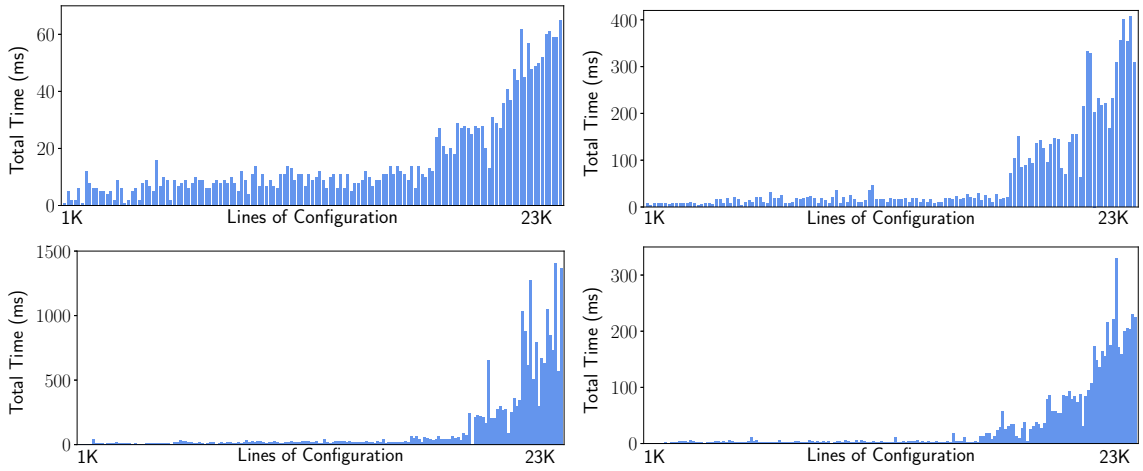


Figure 3.14: Verification time for management interface reachability (upper left), local equivalence (upper right), blackholes (lower left), and fault-invariance (lower right) for real configurations sorted by total lines of configuration.

Real configurations: We benchmarked the verification time for the networks and properties described above. Figure 3.14 (upper left) shows this time for management-interface reachability for each network that is configured with at least one management interface. The networks are sorted by total lines of configuration, with more complex networks appearing farther right. We see that the checks take anywhere from 2 to 60 ms for every network tested. Figure 3.14 (upper right) shows the verification time for local equivalence among routers in each unique role, for all networks with at least two routers in any particular role. Verification time ranges anywhere from roughly 5 to 400 ms. This check is more expensive than management-interface reachability, in part, because it requires more queries. Finally, the lower row of Figure 3.14 shows the time for verification of the absence of blackholes and fault-invariance queries. Both queries take under a second for most networks. The worst case is under 1.5 seconds. While the networks we studied are small, the sub-second verification times we observe are encouraging. They point to the ability of Minesweeper to verify many real configurations in an acceptable amount of time. Next, we stress test our tool by running it on larger, albeit synthetic networks.

Synthetic configurations: To test the scalability of our tool on larger networks, we use a collection of synthesized, but functional, configurations for data center networks of increasing size. Each

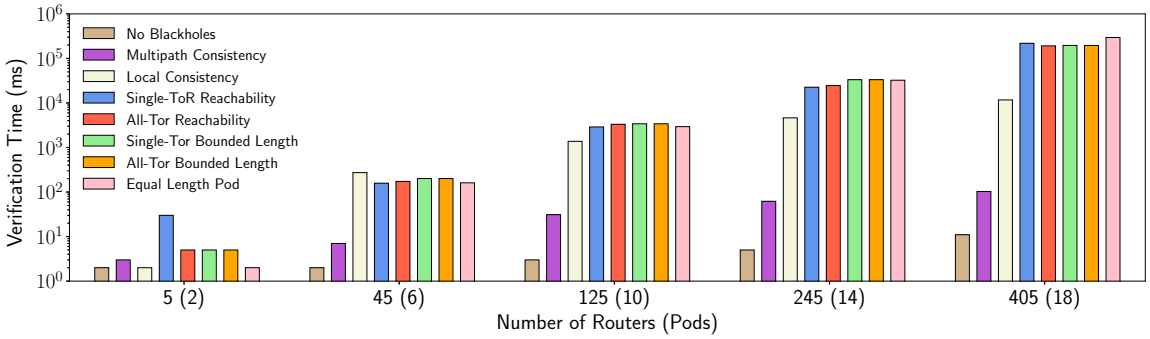


Figure 3.15: Verification time vs. network size for synthetic configurations.

data center uses a folded-Clos topology and runs BGP both inside the network as well as to connect to an external backbone network. Each top-of-rack router in the data center is configured to advertise a /24 prefix corresponding to the shared subnet for its hosts. All routers are configured to enable multipath routing to evenly distribute load across all of its available peers. Spine routers in the data center connect to external neighbors in the adjacent backbone network and are configured to use route filters on all externally connected interfaces to block certain advertisements.

For each network, we use Minesweeper to check a large collection of the properties described in Section 3.7. First, we fix a destination ToR and use queries to check both single-source and all-source reachability from other ToRs. Similarly, we also check that both some and all other ToRs will always use a path to the destination ToR that is bounded by four hops, to ensure that traffic never uses a "valley" path that goes down, up, and then down again. To demonstrate a query that asks about more than a single path, we verify that all ToRs in a separate pod from the destination will always use paths that have equal length. This ensures a certain form of symmetry in routing. In addition to path-based properties, we also verify the multipath-consistency property that every router in the network will never have different forwarding behavior along different paths. We also check that every spine router in the network is equivalent using the local-consistency property. To ensure that all n spine routers are equivalent, we check for local equivalence among pairs using $n - 1$ separate queries. If all routers are equivalent, then transitively they are equivalent as well. Finally, we verify the absence of black holes in the data center.

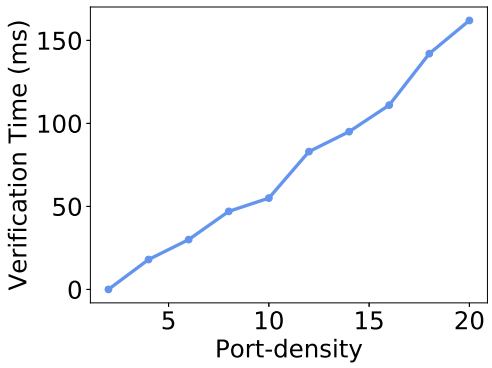


Figure 3.16: Scalability of a single local equivalence check.

Figure 3.15 shows the time to check each property for data centers of different size. Multipath consistency and the no-blackholes properties are the fastest to check, taking under a second to verify in all cases. This speed is in most part due to the minimal use of ACLs in the configurations. The solver quickly determines that the properties cannot be violated because the control and data planes stay in sync. The next fastest property to verify is local equivalence among spine routers. This check takes under 2 minutes for the largest network. In this case, each pairwise equivalence check takes roughly 145 milliseconds. The most expensive properties pertain to reachability and path-length. For the largest network it takes under 5 minutes to verify such properties. Interestingly, queries checking all-source vs single-source take approximately the same amount of time. Instead of checking the property by issuing multiple queries, as is the case in many prior, path-based tools [40, 100], all-source reachability is a single query in our graph-based formulation.

Dissecting Local Equivalence: In contrast with the other properties, checking local equivalence among spine routers requires more than one query. To better understand the complexity of this operation, we look at how a single local-equivalence query scales as a function of the port-density of the data center. Each additional port corresponds to an additional neighbor, which is then modelled as another symbolic environment to the router. Figure 3.16 shows the results. From the graph, verification time appears to scale linearly with the port density.

3.10.3 Optimization Effectiveness

We evaluated the effectiveness of the optimizations described in Section 3.8 by comparing verification time for single-source reachability queries in the synthetic networks both with and without optimizations. The prefix-hoisting optimization that replaces symbolic variables representing an advertised prefix with instances of the global destination IP variable has a large impact on performance, speeding up verification by over 200x on average. This is due to the fact that bitvectors are expensive for SMT solvers. Solvers typically deal with bitvectors by "bit blasting" them into SAT. However, this introduces 32 additional variables into the model for every edge in the graph. The next two optimizations: merging common import and export attributes of variables and specializing variables by protocol, are both forms of slicing optimizations. Together, these optimizations improve the performance of the solver roughly 2.3x on average over prefix hoisting alone.

3.11 Summary

In this chapter, we presented a general approach to the problem of network control plane verification. We introduced SRPs as a formal model for both a routing protocol as well as the network on which it runs. Stable solutions to the routing problem can be captured in terms of logical constraints, leading to a relatively direct translation from an SRP to an off-the-shelf SMT solver. We demonstrated how, through such a translation, it is possible to check that a wide variety of properties such as reachability and path length hold in every stable data plane that might emerge from the control plane. Through a series of optimizations, we showed how it is possible to scale up such an approach to work with networks consisting of several hundreds of routers.

While this approach to network control plane verification is highly general, there remain several limitations. For one, we saw that several features of the control plane are challenging to encode in logic. For example, Protocols such as iBGP and static routes that require recursive lookup introduce cross-destination dependencies that require encoding duplication, leading to a larger SMT problem and a corresponding degradation of verification performance. Another limitation is

that the scalability verification can grow exponentially with the network size, even for very simple and structured networks. In the next section, we attempt to address this problem of scalability.

Chapter 4

Control Plane Verification with Abstraction

We have seen how network verification can be done in a practical and general way in Chapter 3. However, scaling verification to many of the largest networks in practice remains a challenging problem. For instance, from Figure 3.15 we can see that the time it takes to check a single reachability query with Minesweeper grows exponentially with the size of the network, even for relatively simple networks. Furthermore, the challenge of scaling network analysis is not unique to Minesweeper. For example, in Batfish [44], a testing tool, the time it takes to model control plane dynamics limits the number of tests that can be administered. Similarly, the cost of other verification/testing tools often grows exponentially in the worst case, and in practice, tops out at a few hundred devices—far short of the 1000+ devices that are used to operate many modern data centers.

In this Chapter, we address the problem of scalability by defining a new theory of control plane equivalence in terms of the SRP formalism. Using this theory, we can compress large, concrete networks into smaller, abstract networks with equivalent control plane behavior. Because the compression techniques we present preserve many properties of the network control plane—including reachability, path length, loop freedom, and convergence—analysis tools of all kinds can operate quickly on the smaller networks, rather than their larger concrete counterparts. In

other words, this theory is an effective complement to ongoing work on network analysis, capable of helping accelerate a wide variety of analysis tools.

Intuitively, the reason it is possible to compress control planes in this fashion is that large networks tend to contain quite a bit of structural symmetry—if not, they would be even harder to manage by humans. For instance, many spine (or leaf or aggregation) routers in a data center may be configured similarly.

4.1 Related Work

The idea of leveraging the inherent symmetries in programs and problem domains that arise in practice to scale analysis has been studied before, both in the context of software verification and in networks. Here we provide a brief overview:

Abstractions in verification: Conservative abstractions are the mainstay of program verification in various forms such as loop invariants [43, 62], abstract interpretation [33], and counterexample guided abstraction refinement [8, 28, 29]. These abstractions enable sound analysis for verification problems that are often undecidable or intractable. Tighter abstractions based on symmetry and bisimulations have also been used successfully to scale model checking [27, 38]. We build on these foundations to seek useful abstractions for compressing networks that preserve control plane equivalence.

Abstractions in networks: Recently, work [84] exploited the intuition of symmetry to scale verification. However, this work operates over the (stateless) network data plane, *i.e.*, the packet-forwarding rules, as opposed to the control plane, *i.e.*, the protocols that distribute the available routes. While both the data and control planes process messages (data packets and routing messages, respectively), the routing messages interact with one another whereas the data packets do not. More specifically, data packet processing depends only on the static packet-forwarding rules of a router; it does not depend on other data packets. In contrast, routing messages interact: the

presence and timing of one message can cause another message to be ignored. Such interactions create dynamics not present in stateless data planes and can even lead to many different routing solutions for the same network. Consequently, we face new and different set of challenges from this earlier work: our formulation of control plane semantics, the form of network abstractions, the properties preserved, and the inference algorithms are all entirely different.

One interesting prior work has explored the role of *control plane* symmetry reductions, specifically for the BGP routing protocol [99]. This work focuses primarily on using symmetry to preserve convergence properties of BGP by using local topology rewrites based on router configurations. In contrast, this thesis focuses on a notion of control-plane equivalence, which includes preserving a wide variety of properties such as reachability and path length, in addition to convergence. In addition this chapter describes how to automatically extract symmetries for an arbitrary SRP, which includes a wide variety of protocols like OSPF, RIP, and static routing, in addition to BGP. The notion of local rewrite in this earlier work is similar to a notion of an effective abstraction that we develop in this chapter, however, effective abstractions can include non-local rewrites (*e.g.*, for BGP) that are not possible in this previous work.

4.2 Overview

Our goal is to define an algorithm that, given one SRP, computes a new, smaller SRP that exhibits "similar" control plane behavior. We call the input SRP the *concrete network*, and the output SRP the *abstract network*. A *network abstraction* defines precisely the relationship between the two. It is a pair of functions (f, h), where f is a *topology abstraction* that maps the nodes and edges of the concrete network to those of the abstract network, and h is an *attribute abstraction* that maps the concrete attributes in control plane messages to abstract ones. Two networks are *control-plane equivalent (CP-equivalent)* when:

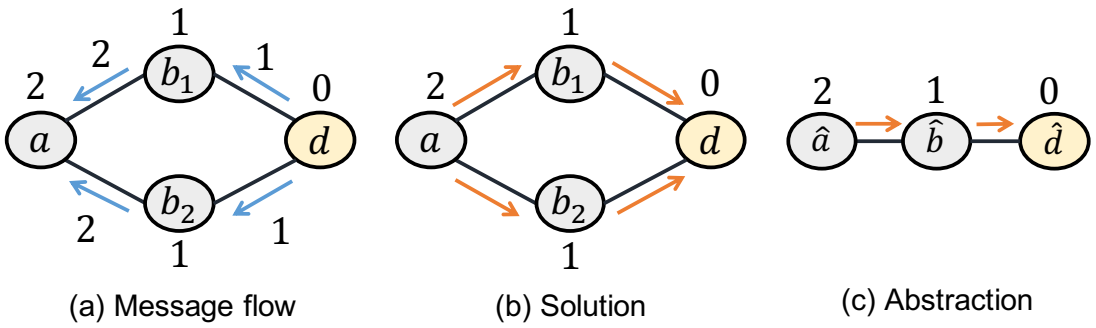


Figure 4.1: Network running RIP and its abstraction.

There is a solution $\mathcal{L}$ to the concrete network iff there is a solution $\widehat{\mathcal{L}}$ to the abstract network where (i) routers are labeled with similar attributes, as related by the attribute abstraction; and (ii) packets are forwarded similarly, as related by the topology abstraction.

CP-equivalence is powerful because it preserves many properties such as reachability, loop-freedom, and convergence. Moreover, because the connection between abstract and concrete networks is tight (*i.e.*, a bisimulation) as opposed to an over-approximation, bugs found when verifying the abstract network, correspond to real bugs in the concrete network (*i.e.*, no false positives). Likewise, because the abstractions are not under-approximations, if we verify that there are no violations of a property in the abstract network, then there are no violations of the property in the concrete network (*i.e.*, no false negatives).

Example 1: Figure 4.1(c) shows a CP-equivalent abstraction of an example network running RIP. Recall that RIP passes messages based on hop count to the destination. The (unique) solution to the RIP SRP instance is shown in Figure 4.1(b) with the forwarding behavior shown with the arrows. The abstract network (c) is a smaller network that collapses b_1 and b_2 into a single node $\hat{b}$. More specifically, the topology abstraction f maps the concrete node a to $\widehat{a}$, b_1 and b_2 to $\widehat{b}$, and d to $\widehat{d}$, while the attribute abstraction h is simply the identity function, leaving hop count unchanged. The abstraction is CP-equivalent because there is only one stable solution to both abstract and concrete networks, and given a concrete node n, the label associated with that node is the same as the label associated with $f(n)$. For instance, b_1 is labeled with attribute 1 and so is $\widehat{b}$, its corresponding node

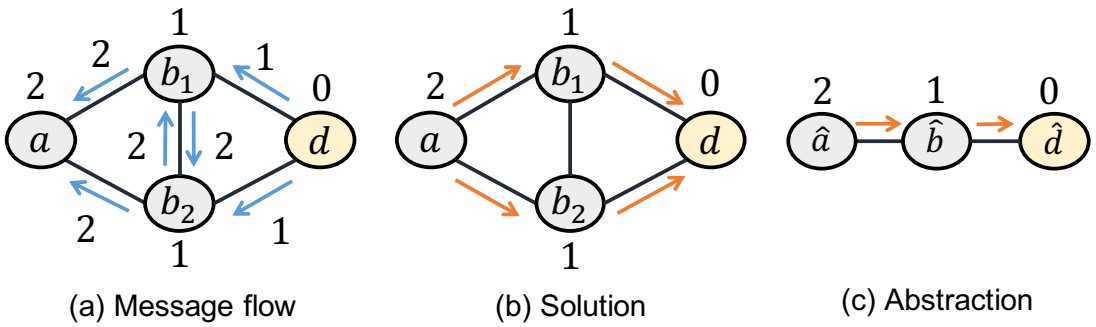


Figure 4.2: Network from Figure 4.1 with the middle edge added.

in the abstraction. One can also observe that the forwarding relation in the concrete network is equivalent (modulo f) to the forwarding relation in the abstract network. For instance, concrete node b_1 forwards to d and the corresponding abstract node $\widehat{b}$ forwards to $\widehat{d}$ as well.

Example 2: Figure 4.2 shows the same example from before, but now with an edge added between b_1 and b_2. As before, there is a single unique solution to both the concrete and abstract networks, and their solutions are in one-to-one correspondence. Hence, the abstraction in (c) is a valid abstraction for this network. However, it is only after computing the stable solutions for the networks that we can see that the new edge between b_1 and b_2 is never used in a solution. In general, if b_1 forwarded through b_2, then the abstraction in (c) would not be able to capture this behavior. This leads to the following idea.

Effective Abstractions: While CP-equivalence is our goal, we cannot evaluate pairs of networks for equivalence directly—naively, one would have to simulate the behavior of the pair of networks on all possible inputs, an infeasible task. Instead, we formulate a set of conditions on network abstractions that imply CP-equivalence and can be evaluated efficiently. *Effective abstractions* are those that satisfy these conditions. Because effective abstractions are sufficient (but not necessary) conditions for CP-equivalence, there are some abstractions that are possible, but we will miss. In particular, effective abstractions are sufficient to find the abstraction for the network in Figure 4.1, but not for the network in Figure 4.2.

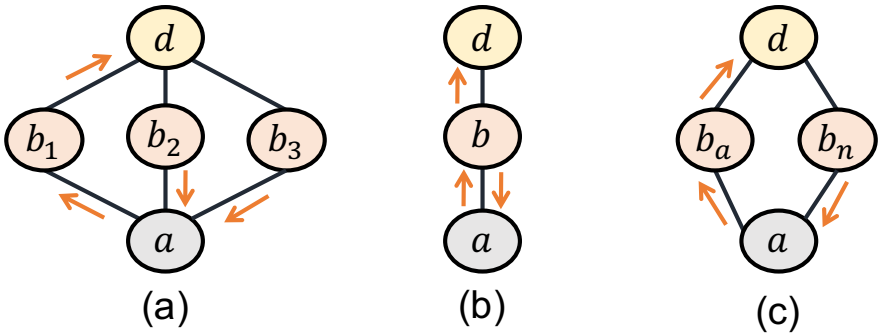


Figure 4.3: Example abstraction for BGP: (a) Concrete BGP network. (b) Unsound abstraction (has a loop). (c) Sound abstraction.

While these conditions help us identify abstractions for protocols such as RIP and OSPF, there is a serious complication for BGP. One of the conditions is *transfer-equivalence*, *i.e.,* the routing information is transformed in a similar way in concrete and abstract networks. However, BGP routers employ an implicit loop-prevention mechanism that rejects routes that contain their own AS (Autonomous System, an identifier for the network) number. Consequently, even when two routers have identical configurations, their transfer functions are slightly different because they reject different paths.

To handle this complication, we define an extended set of conditions, called *BGP-effective conditions*. These conditions can also imply CP-equivalence and can be evaluated efficiently, though the relationship between abstract and concrete networks is more sophisticated; the function mapping nodes in the concrete to the abstract networks is not fixed but instead depends on the particular solution to which the control plane converges.

More precisely, given a concrete SRP and an effective abstraction, which produces $\widehat{SRP}$, a BGP-effective abstraction provides an intermediate network $\overline{SRP}$. This intermediate network is similar to $\widehat{SRP}$ except that an abstract node $\widehat{n}$ in $\widehat{SRP}$ is split into several nodes—one for each possible forwarding behavior of $\widehat{n}$. Importantly, we prove that the number of instances node $\widehat{n}$ needs to be split into, is bounded by k, where k is the number of different BGP local preference values that the concrete nodes may use.

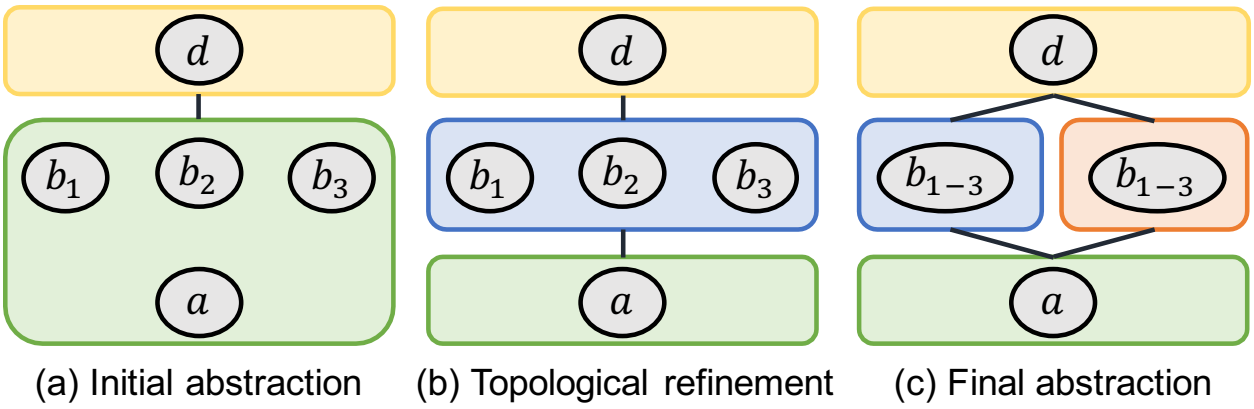


Figure 4.4: Abstraction refinement for the network in Figure 4.3(a). Boxes represent abstract nodes.

Figure 4.3 shows a situation in which these sorts of difficulties arise. Assume the middle routers (b_1, b_2, b_3) of the concrete network have identical configurations and prefer to route traffic down rather than up. Despite this preference, one of the three must route upwards. In the figure, b_1 happens to be that router. This solution is stable—no router receives a route from a neighbor that it prefers to the current route (if router b_1 were to receive a route from a, the path to d would be $b_1.a.b_1.d$, a loop which b_1 would reject). And yet, despite identical configurations, routers b_1 and b_2 forward in different directions. Figure 4.3(b) shows a naive (and incorrect) abstraction in which all three of b_1, b_2 and b_3 are collapsed to the same node. This abstract network in (b) is not CP-equivalent to the network in (a), because mapping the solution to (a) in (b) requires generating a forwarding loop. However, there does exist a smaller CP-equivalent abstract network—the network depicted in Figure 4.3(c). The latter network is capable of mapping the solution depicted in Figure 4.3(a) without introducing a forwarding loop.

From Theory to Practice: Our theory provides the basis for developing an efficient algorithm for control plane compression. Based on *abstraction refinement*, our algorithm first generates the coarsest possible abstraction and then repeatedly splits abstract nodes until the resulting network satisfies the conditions of a (BGP-)effective abstraction.

Figure 4.4 visualizes the algorithm on the BGP network of Figure 4.3(a). As a first step in Figure 4.4(a), we generate the coarsest possible abstraction: the destination is represented alone as one abstract node and all other nodes are grouped in a separate abstract node. This first abstraction is not an effective abstraction—it does not satisfy a topological condition requiring that all concrete nodes (b_1, b_2, b_3, a) associated with one abstract node have edges to some concrete node (d) in an adjacent abstract node. In this case, concrete node a does not satisfy the condition. It is thus necessary to refine the abstraction by separating nodes b_1, b_2, and b_3 from a.

Figure 4.4(b) presents the second refinement step, where the topological condition is satisfied but the BGP-effective conditions are not: The nodes b_1, b_2, and b_3 use one non-default BGP local preference to prefer routing down rather than up and as a consequence each node may exhibit up to two possible behaviors. Consequently, we must split the abstract node for b_1, b_2, and b_3 into two separate nodes. We do not know statically the mapping of concrete to abstract nodes, so our visualization places all three concrete nodes in each abstract node to represent all possible mappings.

Figure 4.4(c) happens to satisfy all conditions of a BGP-effective abstraction. Consequently, the refinement process terminates. The final abstraction includes 4 abstract nodes and 4 total edges—a reduction in size from our concrete network with 5 nodes and 6 edges. Although this simple example does not show much reduction, as we show later, significant reductions are possible in larger networks.

4.3 Abstraction Definitions

Intuitively, a network abstraction is a transformation that relates two SRPs—a concrete $SRP = (G, a_d, A, \prec, \mathsf{trans})$ and an abstract $\widehat{SRP} = (\widehat{G}, \widehat{a_d}, \widehat{A}, \widehat{\prec}, \widehat{\mathsf{trans}})$—using a pair of functions (f, h). The topology function $f : V \to \widehat{V}$ maps each concrete graph node to an abstract graph node, and the attribute function $h : A \to \widehat{A}$ maps each concrete attribute to an abstract one. For convenience, we will write $u \mapsto \widehat{u}$ to mean $f(u) = \widehat{u}$, and $a \mapsto \widehat{a}$ to mean $h(a) = \widehat{a}$. We also freely apply f

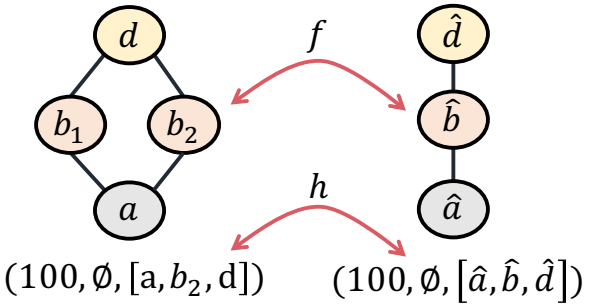


Figure 4.5: Example attribute abstraction function for BGP.

to edges and paths: given an edge $e = (u, v)$, $f(e)$ means $(f(u), f(v))$; given a path $u_1, \ldots, u_n$, $f(u_1, \ldots, u_n)$ means $f(u_1), \ldots, f(u_n)$.

Attribute abstraction (h) allows the set of attributes to differ between the concrete and abstract networks. This ability may be used, for example, to convert attributes with concrete nodes into those with related abstract nodes. For instance, in the BGP network in Figure 4.5, f maps b_i nodes to the abstract node $\widehat{b}$, while h maps the concrete AS path to its abstract counterpart.

4.3.1 Effective Abstraction Conditions

The definition of a network abstraction as a pair of functions, f and h, is highly general and flexible. However, we are primarily interested only in abstractions that preserve the control plane behavior of the concrete network. An *effective* abstraction satisfies a set of relatively easy-to-check conditions that imply CP-equivalence. These conditions, listed in Figure 4.6, are restrictions on the topology function f and the attribute function h.

Well-formed SRPs: The first such effective conditions we refer to as well-formedness conditions. In an SRP, the $\prec$ relation and `trans` function can compare and modify attributes arbitrarily. While this generality helps model a wide variety of routing protocols, it also allows nonsensical behaviors. We define *well-formed* SRPs as those with 3 practical properties: (1) *self-loop-freedom*: The graph must not contain self loops: $\forall v.(v, v) \notin E$. (2) *non-spontaneity*: If a neighbor has no route to the destination, then a router will not obtain a route from that neighbor. While useful, non-

Network abstraction $\boxed{(f,h) : (V \to \widehat{V}) \times (A \to \widehat{A})}$

$SRP = (G, A, a_{\mathrm{d}}, \prec, \mathsf{trans})$ *concrete SRP instance*
$\widehat{SRP} = (\widehat{G}, \widehat{A}, \widehat{a_{\mathrm{d}}}, \widehat{\prec}, \widehat{\mathsf{trans}})$ *abstract SRP instance*

$u \mapsto \widehat{u} \;\equiv\; f(u) = \widehat{u}$ *vertex abstraction notation*
$a \mapsto \widehat{a} \;\equiv\; h(a) = \widehat{a}$ *attribute abstraction notation*

SRP well-formedness

$\forall v.\ (v,v) \notin E$ *self-loop-free*
$\forall e.\ \mathsf{trans}(e, \bot) = \bot$ *non-spontaneous*
$\forall a.\ a \neq \bot \implies a \prec \bot$ *drop-ordering*

Effective abstractions

$(d \mapsto \widehat{d}) \wedge (\forall d'.\ d \neq d' \implies d' \not\mapsto \widehat{d})$ *dest-equivalence*
$h(a_d) = \widehat{a_d}$ *orig-equivalence*
$\forall a, b.\ a \prec b \iff h(a) \mathrel{\widehat{\prec}} h(b)$ *rank-equivalence*
$\forall e, a.\ h(\mathsf{trans}(e,a)) = \widehat{\mathsf{trans}}(f(e), h(a))$ *trans-equivalence*
$\forall u, v.\ (u,v) \in E \implies (\widehat{u}, \widehat{v}) \in \widehat{E}$ $\forall\exists-\mathsf{abstraction1}$
$\forall \widehat{u}, \widehat{v}.\ (\widehat{u}, \widehat{v}) \in \widehat{E} \implies \big(\forall u.\ u \mapsto \widehat{u} \implies \exists v.\ v \mapsto \widehat{v} \wedge (u,v) \in E\big)$ $\forall\exists-\mathsf{abstraction2}$

BGP-effective abstractions

$\forall u, v.\ (u,v) \in E \iff (\widehat{u}, \widehat{v}) \in \widehat{E}$ $\forall\forall-\mathsf{abstraction}$
$\forall e, a.\ e = (u,v) \wedge u \notin a.\mathrm{path} \implies h(\mathsf{trans}(e,a)) = \widehat{\mathsf{trans}}(f(e), h(a))$ *transfer-approx*

CP-equivalence $\boxed{SRP \equiv \widehat{SRP}}$

$\mathcal{L} \in SRP \iff \widehat{\mathcal{L}} \in \widehat{SRP}$ when:
1. $\forall u.\ h(\mathcal{L}(u)) = \widehat{\mathcal{L}}(f(u))$ *label-equivalence*
2. $\forall (u,v) \in E.\ (u,v) \in \mathsf{fwd}_{\mathcal{L}}(u) \iff (\widehat{u}, \widehat{v}) \in \widehat{\mathsf{fwd}}_{\widehat{\mathcal{L}}}(\widehat{u})$ *fwd-equivalence*

Figure 4.6: Definitions for SRP abstractions and abstraction properties.

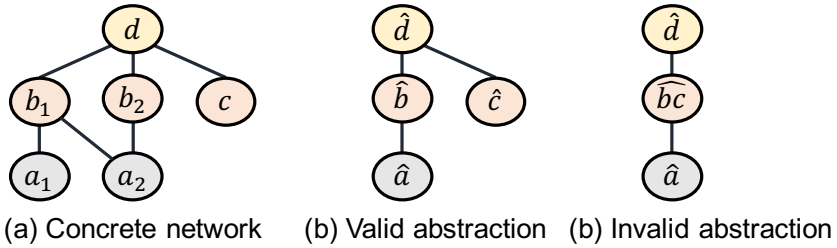


Figure 4.7: Valid and invalid topology abstractions.

spontaneity is not necessary for all of our theoretical results (e.g., see SRPs for static routing). (3) *drop-ordering*: The special value $\perp$ is the "worst" attribute value. We typically get this property for free by construction by adding $\perp$ to the user-provided SRP definition.

Topology abstraction conditions: Effective topology functions obey two conditions. First, they preserve the identity of the destination node (*dest-equivalence*). That is, the concrete destination node, and only this node, should be mapped to the abstract destination: $d \mapsto \widehat{d}$, $d' \not\mapsto \widehat{d}$. Second, the topological mapping as a whole must be what we call a forall-exists abstraction (an abstraction satisfying $\forall\exists-$abstraction1 and $\forall\exists-$abstraction2). A $\forall\exists-$abstraction demands that we can not allow there to be some concrete edges, but no abstract edge. Thus for all concrete edges (u, v) in the concrete network there must be a corresponding abstract edge $(\widehat{u}, \widehat{v})$ ($\forall\exists-$abstraction1). Similarly for every ($\widehat{u}$, $\widehat{v}$) edge in the abstract network, *all* concrete nodes u that map to $\widehat{u}$ must have an edge to *some* other concrete node v that maps to $\widehat{v}$ ($\forall\exists-$abstraction2).

Figure 4.7 shows an example of both a valid and invalid $\forall\exists-$abstraction. The abstraction on the right is invalid because c does not have an edge to either a_1 or a_2 despite there being an edge between $\widehat{bc}$ and $\widehat{a}$ in the abstract network. For example, for the concrete network in Figure 4.7(a), the abstract network in 4.7(b), which maps b_i nodes to $\widehat{b}$ and c_1 to $\widehat{c}$, is a valid $\forall\exists-$abstraction; for the abstract edge from $\widehat{a}$ to $\widehat{b}$, each concrete node a_i has an edge to some concrete node b_i. On the other hand, also mapping both b_i nodes and c_1 to a single abstract node $\widehat{bc}$, as in Figure 4.7(c), will violate $\forall\exists-$abstraction conditions because there would be an abstract edge between $\widehat{a}$ and $\widehat{b}$, but neither a_1 nor a_2 has an edge with c_1. Similarly, mapping both b_1 and a_1 to the same abstract node

would also be invalid because that would create a self-loop in the abstract network, which violates a condition for well-formed SRPs. Thus, while the topological function conditions admit a fair degree of flexibility, they also limit how small the abstract network can be, since it must preserve control plane equivalence.

Attribute abstraction conditions: The final set of conditions required for an effective abstraction involve the attribute abstraction function h. The first condition for attribute abstraction, *orig-equivalence*, states that the abstraction function must preserve the destination attributes: $h(a_d) = \widehat{a_d}$. An abstraction must also preserve the comparison relation's attribute ordering with *rank-equivalence*: $a \prec b \iff h(a) \mathrel{\widehat{\prec}} h(b)$. Finally, an abstraction must preserve the transfer function with *transfer-equivalence*: $h(\mathsf{trans}(e, a)) = \widehat{\mathsf{trans}}(f(e), h(a))$. That is, applying the concrete transfer function and abstracting the resulting attribute should be the same as abstracting the attribute first, and then applying the abstract transfer function. A critical aspect of the transfer-equivalence property is that, unlike CP-equivalence, which is a network-wide property, transfer-equivalence is a property that can be checked efficiently by comparing the transfer functions locally.

4.4 Control Plane Equivalence

We prove that effective abstractions guarantee CP-equivalence in two steps. First, we demonstrate that effective abstractions are *label-equivalent* (Figure 4.6). In other words, for each solution $\mathcal{L}$ to the concrete SRP, there exists a corresponding solution $\widehat{\mathcal{L}}$ to the abstract $\widehat{SRP}$ where $\forall u.\ h(\mathcal{L}(u)) = \widehat{\mathcal{L}}(f(u))$ (*i.e.*, whenever $\mathcal{L}$ labels node u with attribute a, $\widehat{\mathcal{L}}$ labels $f(u)$ with $h(a)$). This also must hold in the other direction (for each solution $\widehat{\mathcal{L}}$, there exists a corresponding solution $\mathcal{L}$). Next, we show that given related labellings, the final control plane behaviors are also related, *i.e.*, they are equivalent with respect to forwarding (*fwd-equivalent* as defined in Figure 4.6). Finally, we show that forwarding equivalence preserves a wide variety of properties, including reachability. Full proofs for CP-equivalence can be found in the Appendix Section A.1.

4.4.1 Loop-free protocols

Our proof depends on the structure of the SRPs and their solutions. In particular, when the SRP nodes dynamically transmit information to one another, we would like to be able to carry out the proof using induction on the depth of the forwarding tree. However, we cannot do that if the SRP solutions contain loops, as the induction would not be well-founded. Fortunately, most broadly-used dynamic routing protocols are loop-free by design. We will first consider the case of loop-free protocols, and then separately consider the simpler case of static routes, which can be configured to create loops.

Definition 4.4.1. *An SRP instance is loop-free if there is no stable solution to the SRP that contains a forwarding loop.*

Most protocols are loop-free by design – *e.g.*, BGP prevents loops using a special loop-prevention mechanism, and OSPF avoids loops by taking advantage of strictly increasing path cost. If a protocol is both loop-free and non-spontaneous ($\mathsf{trans}(e, \bot) = \bot$), then we know something about the shape of its solutions.

Theorem 4.4.1. *The forwarding behavior for any solution $\mathcal{L}$ to a well-formed, loop-free SRP will form a DAG rooted at the destination d.*

Using this property of stable solutions, we can prove that for any concrete solution $\mathcal{L}$, there is an abstract solution $\widehat{\mathcal{L}}$ such that the solutions are label- and fwd-equivalent (and vice-versa). The proof goes in two steps. First, we prune the network to include only edges in $\mathcal{L}$ or $\widehat{\mathcal{L}}$ that are involved in forwarding. Within such subgraphs, we can show by induction on the length of the forwarding paths that the subgraphs satisfy label-equivalence and fwd-equivalence. It is then easy to come to our desired conclusion by showing that adding the removed edges back to the network does not affect the solution of either the concrete or the abstract graph.

Theorem 4.4.2. *A well-formed, loop-free SRP and its effective abstraction $\widehat{SRP}$ are label- and fwd-equivalent.*

Using this theorem we may also conclude that any effective abstractions of common protocols, which produce loop-free routing, are CP-equivalent. However, effectiveness requires transfer-equivalence, which as mentioned previously commonly does not hold for BGP. That makes it impossible to obtain effective abstractions for BGP networks. We will address this shortcoming shortly by defining another kind of abstraction that is applicable for BGP.

4.4.2 Static routing

Networks with static routes are not necessarily loop-free. (The presence of a loop would clearly be a bug, but we must be sure our theory is sound in such a situation so we can use it to detect inadvertent bugs caused by misconfiguration of static routes). Fortunately, due to the simple nature of static routing—static routes do not depend on other routes learned from neighbors—we can prove its correctness independently.

Theorem 4.4.3. *Given self-loop-free SRP and $\widehat{SRP}$ for static routing with an effective abstraction, then it is fwd-equivalent.*

4.4.3 Forwarding path equivalence

Next, we lift CP-equivalence to properties of forwarding paths.

Corollary 4.4.4. *Suppose we have a self-loop-free SRP and $\widehat{SRP}$ for RIP, OSPF, static routing, or BGP (without loop prevention), related by effective abstraction (f, h). There is a solution $\mathcal{L}$, where each node $u_1 \mapsto \widehat{u_1}$ forwards along label path $s = \mathcal{L}(u_1) \ldots \mathcal{L}(u_k)$ to some node $u_k \mapsto \widehat{u_k}$ iff there is a solution $\widehat{\mathcal{L}}$ that forwards along the label path $\widehat{s} = \mathcal{L}(\widehat{u_1}) \ldots \mathcal{L}(\widehat{u_k})$ and $h(s) = \widehat{s}$.*

The corollary lifts the property of forwarding equivalence (which relates what neighbors nodes forward traffic to) to forwarding for network-wide paths. Specifically, it relates paths of node *labels*. We use labels rather than nodes so that we can relate properties both of the data plane (forwarding) as well as the control plane (labels). Note that labels are strictly more general as it is

always possible to include the neighbor through which a route is learned in the attribute itself (*i.e.*, by adding a next-hop attribute field).

4.4.4 BGP with Loop Prevention

We model BGP using an abstraction: $h((\mathrm{lp}, \mathrm{tags}, \mathrm{path})) = (\mathrm{lp}, \mathrm{tags}, f(\mathrm{path}))$. Recall that $f(\mathrm{path})$ applies f pointwise over the path. BGP's loop-prevention is problematic here because it depends on the actual concrete path used, which implies that two concrete nodes x and y with syntactically identical configurations will actually have different transfer functions and violate transfer-equivalence. Node x will reject paths that have gone through x but not y, and node y will reject paths that have gone through y but not x. If we were somehow able to abstract away loop prevention, we could attempt to have topology abstractions for BGP that are transfer-equivalent. This observation motivates the additional properties laid out for BGP in Figure 4.6.

BGP-effective abstractions: For BGP, we require dest-, orig- and rank-equivalence as for ordinary effective abstractions. However, as opposed to a $\forall\exists-\mathsf{abstraction}$, we require a slightly stronger (forall-forall) $\forall\forall-\mathsf{abstraction}$. This constraint requires that whenever there is an abstract edge between $\widehat{u}$ and $\widehat{v}$, *all* concrete nodes u that map to $\widehat{u}$ must have an edge to *all* concrete nodes v that map to $\widehat{v}$. This strong condition on the network topology allows us to get away with a weaker condition than transfer-equivalence: we relax the transfer-equivalence condition to what we call transfer-approx. The latter condition is similar to transfer equivalence, except it ignores differences caused by BGP loop-prevention. Formally, it is specified as follows:

$$\forall e, a.\ e = (u, v) \wedge u \notin a.\mathrm{path} \implies h(\mathsf{trans}(e, a)) = \widehat{\mathsf{trans}}(f(e), h(a))$$

In other words, for all edges e and attributes a being advertised from v to u, the concrete and abstract transfer functions are transfer-equivalent if the BGP advertisement (a) does not already contain u in the BGP AS-path.

Bounded behaviors: Now, given a BGP-effective abstraction, we know that, when loop-prevention happens, there may be differences between the forwarding behaviors of different concrete nodes even when they have identical configurations. Fortunately, we can bound the number of different behaviors that can arise dynamically, and, moreover, we can infer that bound directly from the configurations.

Let $\mathcal{B}_{\mathcal{L}}(\widehat{u})$ be the set of possible behaviors of concrete nodes related to abstract node $\widehat{u}$. We can define behaviors as follows:

Definition 4.4.2 (Abstract behaviors)**.** *Given SRP and $\widehat{SRP}$, related by a BGP-effective abstraction, and a concrete solution $\mathcal{L}$, which implies a forwarding relation ($\mathsf{fwd}_{\mathcal{L}}$), the set of behaviors of an abstract node $\widehat{u}$ is:*

$$\mathcal{B}_{\mathcal{L}}(\widehat{u}) = \{\widehat{v} \mid u \mapsto \widehat{u},\ v \mapsto \widehat{v},\ (u,v) \in \mathsf{fwd}_{\mathcal{L}}(u)\}$$

That is, the set of different concrete behaviors for abstract node $\widehat{u}$ is the set of other abstract roles $\widehat{v}$, such that some concrete node in $\widehat{u}$ forwards to some concrete node in $\widehat{v}$.

Next, let $\mathsf{prefs}(v)$ be the set of BGP local-preference values that may be assigned to an announcement at node v. For example, if a configuration explicitly sets the local-preference value to 200 or 300 depending on the route, and 100 is the default local preference, then the set $\mathsf{prefs}(v) = \{100, 200, 300\}$. With these definitions in hand, we have the following theorem.

Theorem 4.4.5. *If a well-formed SRP and $\widehat{SRP}$ for BGP has an $\forall\forall-\mathsf{abstraction}$ and is transfer-approx, then for all solutions $\mathcal{L}$ to SRP, and for all abstract nodes $\widehat{u} \in \widehat{V}$, $|\mathcal{B}_{\mathcal{L}}(\widehat{u})| \leq |\mathsf{prefs}(\widehat{u})|$.*

We give an intuition for the proof here using the example in Figure 4.8. Suppose that u_1 decides to forward to v_{11}. It must be the case that $\mathcal{L}(v_{11})$ is the best choice at u_1 after applying the transfer function from v_{11}. However, because of the $\forall\forall-\mathsf{abstraction}$, every other node u_i has an edge to v_{11} (as well as every other neighbor of u_1), and due to transfer-approx, they will have the same set of choices available to them after applying their transfer functions modulo any dropping due to loop prevention. Therefore, they will all make the same decision as u_1 unless they are already

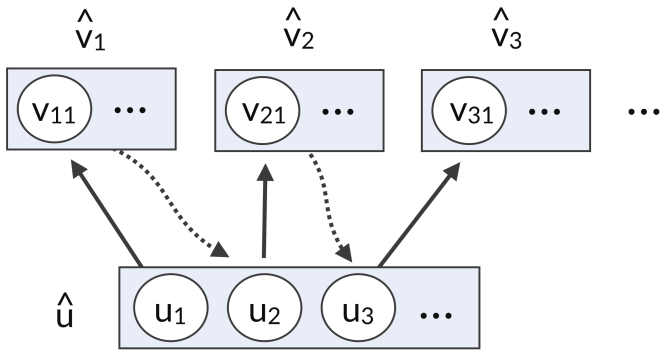


Figure 4.8: A stable solution with the maximum number of behaviors.

on the path used by v_{11} (and thus the option is dropped). In the case that they are already on the path, loop prevention will force them to chose their next best hop. For example, in the Figure v_{11} uses some path that goes through u_2. Thus u_2 is stuck going through its next best path v_{21} and so on with u_3. To get this different behavior at u_1 and u_2 and u_3, it must be the case that the routers u_i set a higher local preference for $\widehat{v_1}$ over $\widehat{v_2}$ over $\widehat{v_3}$ because otherwise they would have simply preferred the shorter path (*e.g.*, u_1 and u_2 would just use v_{31}).

Abstraction refinement: A bound on the number of behaviors for nodes in BGP lets us refine an abstraction by splitting apart abstract nodes into enough cases to recover CP-equivalence. This is similar to the idea of predicate abstraction used successfully in software verification [8, 29], where certain predicates are tracked in the program, and concrete states with identical predicate values map to the same abstract state. In our context, the "predicates" of concern are the forwarding behaviors, such that different forward behaviors should result in different abstract states. We now formalize this intuition.

Suppose we are given an $SRP = (G, A, a_d, \prec, \mathsf{trans})$ for BGP and its abstract version $\widehat{SRP} = (\widehat{G}, \widehat{A}, \widehat{a_d}, \widehat{\prec}, \widehat{\mathsf{trans}})$, which are well-formed and created from a $\forall\forall-\mathsf{abstraction}$ (f, h). We define a new abstraction $\overline{SRP} = (\overline{G}, \overline{A}, \overline{a_d}, \overline{\prec}, \overline{\mathsf{trans}})$ obtained by splitting up each node $\widehat{v}$ into $|\mathsf{prefs}(\widehat{v})|$ copies of the node. We can view the mapping from SRP to $\widehat{SRP}$ as the composition of two abstractions (f_r, h_r) from SRP to $\overline{SRP}$, and (f_s, h_s) from $\overline{SRP}$ to $\widehat{SRP}$, where the comparison

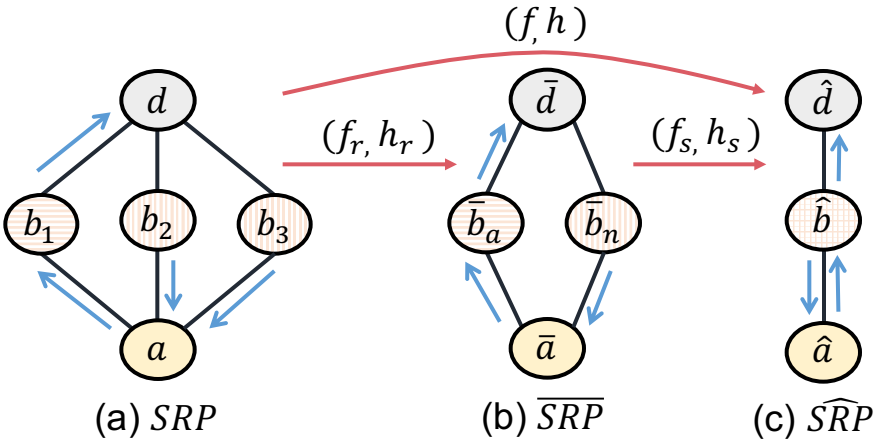


Figure 4.9: Abstraction refinement for Figure 4.3(a).

and transfer functions for $\overline{SRP}$ are copied from $\widehat{SRP}$. Given a new abstraction (f_r, h_r) where $f_r : V \to \overline{V}$ and $h_r : A \to \overline{A}$, we say (f_r, h_r) *refines* (f, h), written as $(f_r, h_r) \sqsubseteq_{(f_s,h_s)} (f, h)$ if f_r is an *onto* function, and $f = f_s \circ f_r$ and $h = h_s \circ h_r$.

We now show that there is a bisimulation between the solutions $\mathcal{L}$ and $\overline{\mathcal{L}}$ as before. However, whereas the abstraction mapping f was known in advance, the refined mapping f_r may change depending on the particular solution $\mathcal{L}$. For example, Figure 4.9(a) shows one of three possible forwarding behaviors for the network. As discussed earlier, with a different message arrival timing, other solutions would have emerged. Depending upon this solution, different nodes, *e.g.*$\{b_1, b_2\}$ or $\{b_1, b_3\}$ would be mapped to $\overline{b_n}$. We do not know which concrete nodes are mapped to which abstract nodes, but we do know that the abstraction has sufficiently many nodes to characterize all possible behaviors.

Theorem 4.4.6. *Suppose we have well-formed SRP, $\widehat{SRP}$, and $\overline{SRP}$ for BGP with an effective abstraction (f, h). There is a solution $\mathcal{L}$ to SRP iff there is a solution $\overline{\mathcal{L}}$ to $\overline{SRP}$, such that there exists a refinement $(f_r, h_r) \sqsubseteq_{(f_s,h_s)} (f, h)$ and $\mathcal{L}$ and $\overline{\mathcal{L}}$ are label- and fwd-equivalent with respect to (f_r, h_r).*

Finally, we have the refinement analogue to forwarding path equivalence:

Corollary 4.4.7. *Suppose we have well-formed SRP, $\widehat{SRP}$, and $\overline{SRP}$ for BGP with an effective abstraction (f, h). There is a solution $\mathcal{L}$, where each node $u_1 \mapsto \widehat{u_1}$ forwards along path $s = \mathcal{L}(u_1) \ldots \mathcal{L}(u_k)$ to some node $u_k \mapsto \widehat{u_k}$ iff there is a solution $\overline{\mathcal{L}}$ where each node $\overline{u_1} \mapsto \widehat{u_1}$ forwards along path $\overline{s} = \mathcal{L}(\overline{u_1}) \ldots \mathcal{L}(\overline{u_k})$ to some $\overline{u_k} \mapsto \widehat{u_k}$ such that $h(s) = h_s(\overline{s})$.*

The key difference between this theorem and the non-BGP case is that the forwarding paths between the concrete network (SRP) and the refined network ($\overline{SRP}$) are equivalent with respect to the nodes they map to in the original abstract network ($\widehat{SRP}$). For example, in Figure 4.9(a), if we want to check that b_2 and b_3 forward along a path that satisfies some property p, then we can not check it against only $\overline{b_a}$ in Figure 4.9(b). Rather, we have to check it against $\overline{b_n}$ as well because there is another stable solution where the roles of b_a and b_n are reversed.

4.4.5 Properties preserved

The CP-equivalence theorems are powerful because effective abstractions preserve labelling and forwarding. Consequently, any property of paths of attributes that holds on the abstract network holds on the concrete network, and any violation in the abstract network is a true violation in some concrete solution. More precisely, consider any predicate P such that $P(s)$ iff $P(h(s))$ for all concrete paths s. There exists a path s such that $P(s)$ in every solution $\mathcal{L}$ to the concrete network iff there exists a path $f(s)$ such that $P(h(s))$ in every solution $\widehat{\mathcal{L}}$ to the abstract network.

Concretely, network operators can verify any of the following properties on small abstract networks and be sure the concrete counterpart satisfies the property as well.

- **Reachability:** Abstract node $f(u)$ can reach $f(v)$ in the abstract network iff concrete node u can reach v in the concrete network.
- **Isolation:** Conversely, abstract node $f(u)$ can *not* reach $f(v)$ in the abstract network iff concrete node u can *not* reach v in the concrete network.
- **Path Length:** All paths between $f(u)$ and $f(v)$ in the abstract network have length n iff all paths between u and v have length n in the concrete network.

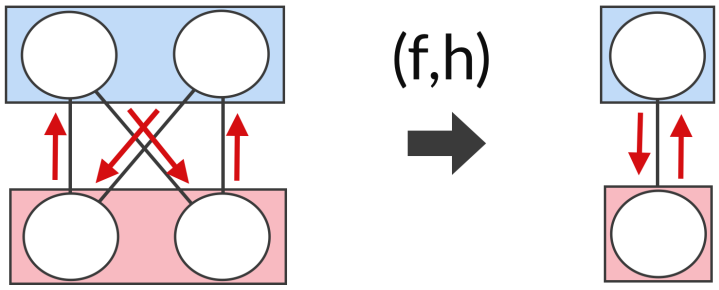


Figure 4.10: Routing loops are preserved under abstraction.

- **Black Holes:** There is no path in the abstract network that ends with the label $\perp$ iff there is no path in the concrete network that ends with the label $\perp$.

- **Multipath Consistency:** Traffic sent from $f(u)$ is reachable along some path to $f(v)$ but dropped along another path iff traffic sent from u is reachable along some path to v and dropped along another path.

- **Waypointing:** Traffic will go through at least one of $\{f(w_1), \ldots, f(w_n)\}$ in the abstract network iff it will be go through at least one of $\{w_1, \ldots, w_n\}$ in the concrete network.

- **Routing Loops:** Routing loops are an interesting case. If there is an abstract routing loop from V to V, then there is a path from each $v_1 \mapsto V$ to some $v_2 \mapsto V$, but v_1 and v_2 may not be the same node. Figure 4.10 shows an example of this kind of behavior where a loop is set up using static routes. However, because there are a finite number of concrete nodes $v_i \mapsto V$, and because each such v_i must eventually reach back to another node v_i', eventually there will be a concrete routing loop. Therefore, there can be a routing loop in the abstract network iff there can be one in the concrete network. However, note that the routing loops are not necessarily of the same length.

- **Control plane announcements:** There are a wide variety of properties one might wish to verify about the flow of control plane announcements. For instance, one might want ensure that announcements tagged with a certain community do not leave the operator's network. This amounts to checking for the existence of paths from a node sending an announcement with that

community attached to a node outside our network. There exists such a path in the abstract network iff there exists such a path in the concrete network.

Convergence: The concrete network necessarily diverges (has no stable solution) iff the abstract network necessarily diverges. To see why, suppose the concrete network had no stable solution, but the abstract network had a stable solution. This would violate CP-equivalence, since each abstract solution has a corresponding concrete solution. Similarly, the concrete network can converge (has some stable solution) iff the abstract network can converge. However, CP-equivalence alone does not guarantee that networks that might converge or might diverge, like the naughty gadget in BGP [56], will necessarily reduce to an abstract network that may diverge.

On the other hand, effective abstractions are stronger than (imply) CP-equivalence. We postulate (but have not proven) that an effective abstraction is sufficient to preserve convergence. For example, it would appear that the concrete network will have a dispute wheel [56] (the lack of which is sufficient condition for convergence safety and robustness) iff the abstract network has a dispute wheel (the nodes in concrete network forming a dispute wheel will induce a dispute wheel in their abstract counterpart).

The ultimate consequence of these facts is that one can build an efficient and sound analysis system by composing an algorithm for effective abstraction with an algorithm for network analysis. However, not all properties are preserved by the abstraction.

4.4.6 Properties not preserved

Although effective abstractions preserve the nature of forwarding paths, they do not, in general, preserve the number of paths or the number of neighbors. Indeed, that is the point—effective abstractions usually reduce the number of paths and neighbors to speed analysis. Consequently, we cannot reason faithfully about fault tolerance. In the abstract network, a single link failure may partition a network whereas in the concrete network, there may exist two or more link-disjoint paths

between all pairs of nodes, allowing the concrete network to tolerate any single failure. Likewise, while our abstractions do not necessarily preserve the number of solutions.

4.5 Abstraction Algorithm

Earlier sections described the conditions under which an abstraction will preserve CP-equivalence, but they give no insight into how one might compute such an abstraction. In this section, we describe an algorithm that computes an abstraction directly from a set of router configurations.

4.5.1 Algorithm Overview

The key requirement for computing an effective abstraction is to ensure that we satisfy each required condition in Figure 4.6. Some conditions such as orig-equivalence ($h(a_d) = \widehat{a_d}$), and rank-equivalence ($a \prec b \iff h(a) \mathrel{\widehat{\prec}} h(b)$) depend only on the particular protocol and choice of h. By fixing h in advance for each protocol, we can guarantee that these conditions hold regardless of the configurations. Several other conditions such as dest-equivalence and $\forall\exists-\mathsf{abstraction}$ depend on the topology, but not on any policy embedded in configurations.

Transfer-equivalence: $h(\mathsf{trans}(e, a)) = \widehat{\mathsf{trans}}(f(e), h(a))$ is the only condition that depends on user-defined policy. Suppose two concrete edges e_1 and e_2 are mapped together by the topology function f. We would have $h(\mathsf{trans}(e_1, a)) = \widehat{\mathsf{trans}}(f(e_1), h(a)) = \widehat{\mathsf{trans}}(f(e_2), h(a)) = h(\mathsf{trans}(e_2, a))$. One simple way to ensure that this equality holds is to only combine together nodes with the same transfer function. For instance, $\mathsf{trans}(e_1, a) = \mathsf{trans}(e_2, a)$ would suffice for e_1 and e_2 to map to the same abstract edge.

Based on the observations above, we fix h for a given protocol; our remaining task is to find a suitable f that satisfies the topology abstraction requirements and only maps together edges with equivalent transfer functions (for a given destination d). We find such a function f using an algorithm based on abstraction refinement. We start with the coarsest possible abstraction where there is a single abstract destination node $\widehat{d}$ and one other abstract node for all other concrete

nodes, and then while the abstraction violates the topology requirements (*e.g.*, $\forall\exists-\mathsf{abstraction}$) or the policy requirements (*e.g.*, transfer-equivalence), we refine the abstraction by breaking up the problematic abstract node into multiple abstract nodes.

This process of breaking up problematic abstract nodes one at a time results in a greedy algorithm. The algorithm is guaranteed to terminate—in the worst yielding the identity function as f—and per results in Section 4.3, will yield a valid abstraction. It may not produce the smallest possible abstract network, though as we show in Section 4.8, it achieves a substantial reduction in size.

Configuration preprocessing: For efficiency, before abstraction refinement, we process router configurations in two different ways.

1. *Destination equivalence classes (ECs):* In our theoretical account of routing, each SRP contains a single fixed destination. However, in practice, configurations contain routing information for many destinations simultaneously. Because announcements for (most) destinations do not interact with one another, we can partition the network into equivalence classes based on where destinations are rooted. Each class has a collection of destination IP ranges and destination node(s). This partitioning allows us to build one abstraction per class instead of one per address. To partition the network into equivalence classes, we use a prefix-trie data structure where the leaves of the trie contain a set of destination nodes.

2. *Encoding transfer function using BDDs:* In order to efficiently find all interfaces that have equivalent transfer-functions for a given destination (class), we use Binary Decision Diagrams (BDDs) [23] to represent the routing policy for each interface. BDDs can compactly represent Boolean functions and have the convenient property that each BDD is canonical for the function it represents. Uniqueness of representation means that semantically-equivalent policies reduce to structurally equivalent BDDs, and turns checking equivalence of any two transfer functions into an $O(1)$ operation after their BDDs are constructed.

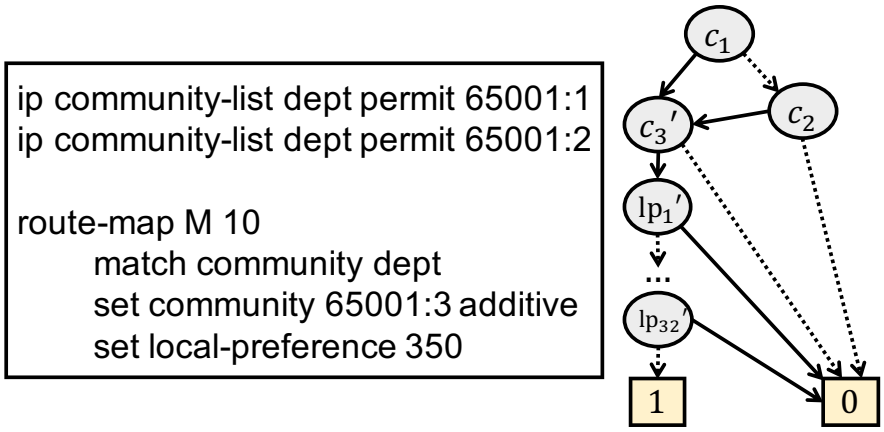


Figure 4.11: BDD for a BGP policy on an interface.

As an example of a BDD encoding, consider the BGP routing policy in Figure 4.11. The policy checks if either the 65001:1 or 65001:2 community is attached to an inbound route advertisement. If so, it adds the 65001:3 community and updates the local preference to 350. Each node in the BDD stands for a boolean variable used to represent state in the advertisement. Primed variables represent output values after applying updates to the advertisement. A solid arrow means the value is true, while a dashed arrow means the value is false. There are two leaf values: 0 and 1 which represent false and true, respectively. Any path from the BDD root to 1 represents a valid input-output relation. If c_1, the variable representing community 65001:1 is true, then the resulting advertisement will have c_3' true (65001:3 attached), and will have a local preference for the 32 bit value representation of 350.

Algorithm 1 Compute abstraction function f

```
1:  procedure FINDABSTRACTION(Graph G, Bdds bdds)
2:      SPECIALIZE(bdds, G.d)
3:      f ← UNIONSPLITFIND(G.V)
4:      SPLIT(f, {G.d})
5:      while True do
6:          V̂ ← PARTITIONS(f)
7:          for û in V̂ do
8:              if |û| ≤ 1 then continue
9:              REFINE(G, bdds, f, û, |prefs(û)|)
10:         V̂′ ← PARTITIONS(f)
11:         if |V̂| = |V̂′| then break
12:     return SPLITINTOBGPCASES(f)
13:
14: procedure REFINE(G, bdds, f, û, numPrefs)
15:     map ← CREATEMAP
16:     for u ∈ û do
17:         for e = (u, v) ∈ G.E do
18:             pol ← GET(bdds, e)
19:             n ← (numPrefs > 1 ? v : f(v))
20:             map[u] ← map[u] ∪ { (pol, n) }
21:     for us ∈ GROUPKEYSBYVALUE(map) do SPLIT(f,us)
```

4.5.2 The Algorithm

Algorithm 1 lists the steps used to compute the abstraction function f given graph G and a collection of BDDs ($bdds$). The first step is to specialize the $bdds$ to the particular destination $G.d$

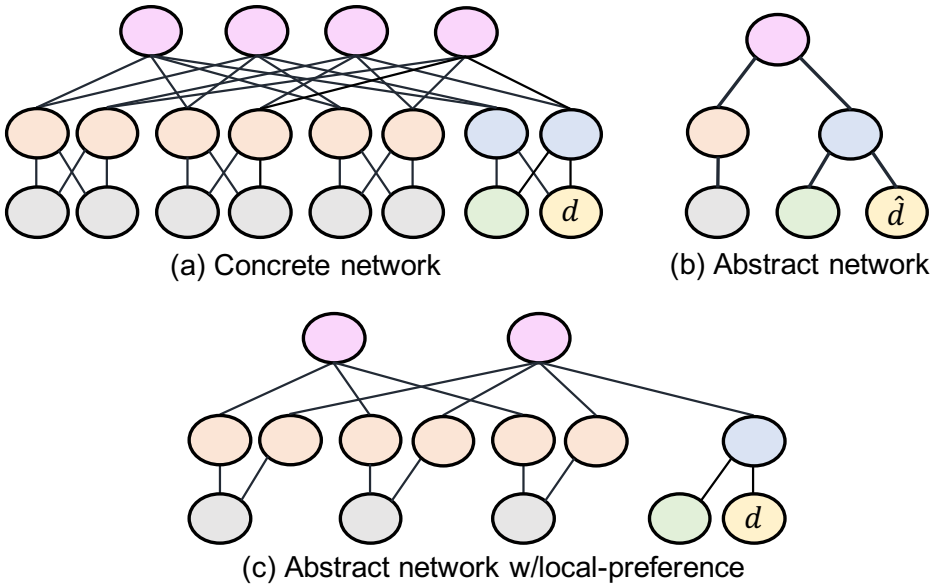


Figure 4.12: Different abstractions for a network running BGP on a fattree topology.

(line 2). We use a union-split-find data structure [96] to maintain a collection of disjoint sets of concrete nodes that represent the abstract nodes in the network. One of the first steps is to split the collection of sets so that $G.d$ becomes its own abstract node (line 4) and every other concrete node remains as a single other abstract node. Next, it repeatedly tries to refine the abstraction while it is not an effective abstraction. The algorithm iterates over each current abstract node. If the abstract node is already fully concrete (line 8), then it continues, otherwise it refines the abstraction. Refine iterates over each concrete node u in $\widehat{u}$ and each edge from u to v, and builds a map from u to a set of pairs of the BDD policy along edge (u, v) and the neighboring node (line 20)—either the concrete neighbor (for $\forall\forall-\mathsf{abstraction}$) or the abstract neighbor (for $\forall\exists-\mathsf{abstraction}$). Finally, we group entries of the map (us) by those values that have the same pairs of policies and neighbors, and then refine by these groups (line 21). This step ensures that groups of devices that have different transfer functions or policies to different neighbors are separated in the next iteration of the algorithm.

Example: Figure 4.12 shows the output of the algorithm on an example with two different routing policies in a fat tree network that uses BGP. In one case, the network uses shortest (AS)

paths routing, and in the second case, the aggregation-layer routers (middle tier) use BGP local-preferences to prefer certain routes over others. The abstract network is bigger in the second case to capture the greater number of possible forwarding behaviors of the aggregation routers.

4.6 Practical Extensions

To apply this abstraction technique to real networks, we address several other commonly used routing features.

Multiple Protocols: Although the stable routing problem is framed in terms of the behavior of a particular protocol, devices in practice often run multiple protocols at once. The problem then is to find an abstraction that is well-formed for every protocol running on each device. To do this, we compute a BDD transfer function for each protocol (OSPF, BGP, etc.) and conceptually, we use a single BDD to represent the combined transfer function of all protocols.

Access Control Lists: In Section 4.4, we showed that the concrete and abstract networks are bisimilar with respect to the final control plane behavior of the network (fwd-equivalent). While ACLs do not affect control plane routing information, they can block traffic from being forwarded out an interface. For this reason, we consider the ACL to be part of the transfer function, which gets captured in the BDD, so that nodes will only be abstracted together if they have the same ACLs with respect to destination d. This ensures that the fwd-equivalence property will not be violated.

iBGP: Recall that iBGP is a complicated protocol that recursively routes packets for eBGP by communicating them over an IGP path. Assuming there is a valid abstraction for both the IGP and for eBGP, and there is no ACL in the network that blocks iBGP loopback addresses, then multiple iBGP neighbors can be compressed together. This is because both iBGP neighbors will receive the same eBGP routes with the same IGP metric. Further, although the iBGP neighbors may have

an edge between them, potentially violating the topology abstraction, this edge is never used since iBGP does not re-advertise routes learned over iBGP to other iBGP neighbors.

External neighbors: A natural consequence of the abstraction algorithm in Section 4.5, is that external (eBGP) neighbors get abstracted as one might expect. For example, if a border router has policy only for three different types of neighbors (*e.g.*, customer, provider, and peer), then a potentially large number of neighbors will get compressed into just 3 abstract neighbors.

4.7 Implementation

We implemented our network abstraction algorithm in a tool called Bonsai. It uses the Batfish [44] network analysis framework to convert network configurations into a vendor-independent intermediate representation. Bonsai operates over this vendor-independent format to create a network abstraction in the form of a smaller, simpler collection of vendor-independent configurations. Tools built using this framework, such as Batfish and Minesweeper, can then work with the smaller configurations to speed up their analysis.

We use the Javabdd [101] library to encode router-level import and export filters, as well as access control lists (ACLs) as BDDs. Because Bonsai creates abstract networks per destination EC, and such ECs are disjoint, our implementation is able to generate abstract networks and check their properties in parallel. We only generate abstract networks for destination ECs that are relevant for a query. For example, checking port-to-port reachability would typically only require generating a single abstract network for one EC.

To integrate Bonsai with Minesweeper, we simply generate abstract networks in parallel, and feed them into Minesweeper, which can check correctness for each EC independently. However, unlike Minesweeper, to answer queries about a single destination class, Batfish must simulate the network for all destinations at once. Therefore, rather than building an abstract network for each destination class, we instead opt to reuse the existing configurations. For each destination class, we select a subset of "canonical" routers in the original network corresponding to the abstract

network. We simply modify the route filters in the configurations so that prefixes overlapping with the destination d are only allowed along edges that appear in the abstraction.

4.8 Evaluation

We evaluate Bonsai using a collection of synthetic and real networks. We aim to answer the following questions: (i) can Bonsai scale to large networks? (ii) can its algorithm effectively compress networks? and (iii) can the abstract, compressed networks be used to speed up network analysis?

Networks studied: We study three types of synthetic network topologies: Fattree [3], Ring, and Full-mesh. Each such network uses eBGP to perform shortest path routing along with destination-based prefix filters to each destination, and its configuration is generated using Propane [14]. These networks are highly symmetric by design and are used to characterize compression as a function of network topology and size. For each topology type, we scale the size and measure the effectiveness and cost of compression.

While the synthetic networks focus on the effect of topology on compression, in practice, most networks do not have perfect symmetry. For this reason, we study operational networks of two different corporations. The first is a datacenter network with 197 routers organized into multiple clusters, each with a Clos-like topology (rather than a single, large Clos-like topology). The network primarily uses eBGP and static routing, with each router running as its own AS using BGP private AS numbers. It also makes extensive use of route filters, ACLs, and BGP communities. All together, it has over 540,000 lines of configuration. Although there are less than 200 routers in the network, there are over 16,000 physical and virtual interfaces in the network.

The second operational network is a wide-area network (WAN) with 1086 devices, which are a mix of routers and switches. The network uses a eBGP, iBGP, OSPF, and static routing, and consists of over 600,000 lines of configuration.

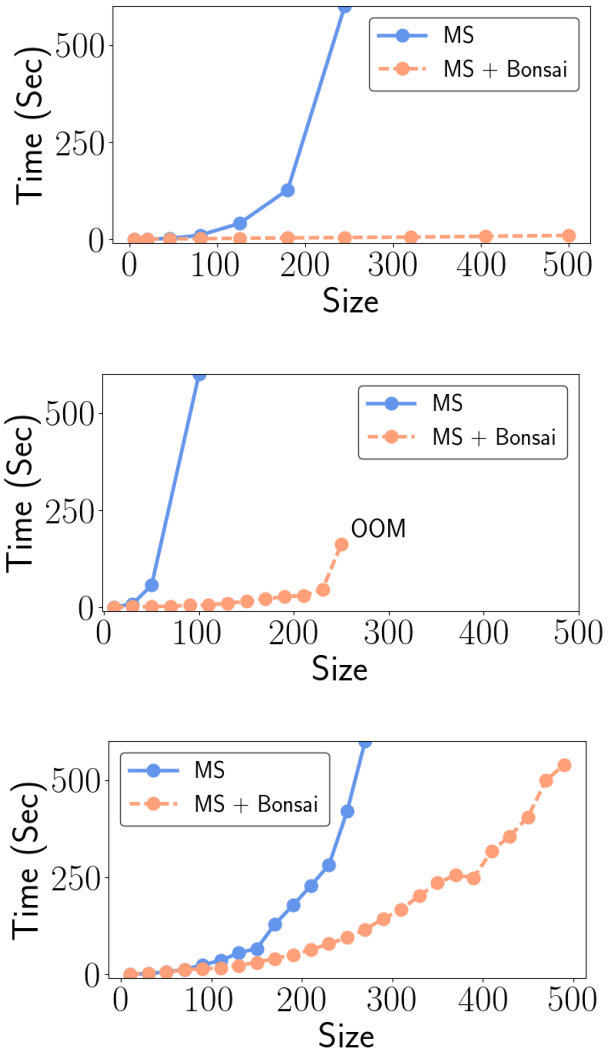


Figure 4.13: Minesweeper (MS) verification time with and without Bonsai for all-pairs reachability.

Topology	V / E	Abs. V / E	Compression ratio	ECs	BDD time	C-time
(a) Synthetic networks						
Fattree	180 / 2124	6 / 5	30× / 424.8×	72	0.36	0.09
	500 / 9100	6 / 5	83.33× / 1820×	200	1.29	0.26
	1125 / 29475	6 / 5	187.5× / 5895×	450	7.87	0.75
Ring	100 / 100	51 / 50	1.96× / 2×	100	0.14	0.08
	500 / 500	251 / 250	1.99× / 2×	500	0.33	2.29
	1000 / 1000	501 / 500	2× / 2×	1000	0.34	12.26
Full Mesh	50 / 1225	2 / 1	25× / 1225×	50	0.18	0.07
	150 / 4950	2 / 1	75× / 4950×	150	1.11	0.34
	250 / 31125	2 / 1	125× / 31125×	250	3.31	5.48
(b) Real networks						
Data center	197 / 16091	30.2 / 143.6 ± 2.2 / ± 18.6	6.6× / 112×	1269	132.28	15.51
WAN	1086 / 5430	209.4 / 759.4 ± 36.5 / ± 129.2	5.2× / 7.2×	845	11.35	1.83

Table 4.1: Compression results for synthetic and real networks. All times are in seconds.

Synthetic network results: Table 4.1(a) shows the results of running Bonsai on the synthetic networks. All experiments were run on an a 8-core Macbook Pro with an Intel i7 processor and 8GB of RAM. The notation V/E stands for the number of nodes and edges in the network respectively. The BDD Time column represents the time it took to compute the BDDs for the network, and the C-time column is the average time it takes to perform compression per equivalence class. For each synthetic network, Bonsai is able to compress the network quickly. For instance, the largest Fattree topology with 1125 nodes takes around 7.9 seconds to build the BDD data structures and an average of .75 seconds per EC to compute the abstract network for the 450 ECs. Because equivalence classes are processed in parallel, it takes under a minute to abstract this network. The compressed network size computed is 6 nodes.

For the Fattree and Full-mesh topologies, the compressed network size stays constant as the concrete network grows. For the ring topology, the compressed network size does grow with the size of the network, and in particular, grows with the diameter of the network. This is necessary since the abstraction must preserve path length. Computing an abstraction for the ring topologies is more expensive because the compression algorithm is only able to split out a single new abstract role with each iteration.

Bonsai's compression has a large effect on network analysis time. Figure 4.13 shows the total verification time to check an all-pairs reachability query compared to topology size for each type of synthetic network using Minesweeper. We use a timeout of 10 minutes. The verification time for abstract networks includes the time used to partition the network, build the BDDs, and compute the compressed network. In all cases, abstraction significantly speeds up verification even when taking into account the time to run Bonsai. Abstracting the Full-mesh topology ran out of memory beyond a certain point, due to the density of the topology.

Real network results: For both networks, we first computed the BDDs and see how many devices have identical transfer functions from their configurations. In the datacenter network, we initially found that there were 112 unique "roles" (set of policies) among the 197 routers. However, many of these differences could be attributed to BGP community values that were attached to routers, but then never matched on in any configuration file. To account for these differences, we use the abstraction function for BGP: $h(\mathrm{lp}, \mathrm{tags}, \mathrm{path}) = (\mathrm{lp}, \mathrm{tags} - \{\mathrm{unused}\}, f(\mathrm{path}))$, which ignores differences from such irrelevant tags. With this abstraction function, we find that there are only 26 unique "roles" among the 197 routers. Further, most of the differences are due to differences in static routes in the configurations. Without static routes, there would only be 8 unique roles. Table 3.14 (b) shows the compression results from this network. It takes just over 2 minutes to compute the BDDs and roughly 15 seconds on average to compute a good abstraction per EC. This time is mainly due to the huge number of virtual interfaces. The average compressed network size is around 30 nodes (a 6.6x reduction), and around 132 edges (a 112x reduction).

For the WAN, we found 137 unique "roles" among the 1086 devices. Many of the differences are from neighbor-specific, prefix-based filters and ACLs. It takes around 11 seconds to compute the BDDs for the network, and under 2 seconds per EC to compute a good abstraction. The average compressed size achieves a 5.2x reduction in the number of nodes and a 7.4x reduction in the number of edges.

Finally, to test the effectiveness of Bonsai at facilitating scalable analysis of real networks, we run a reachability query between two devices in Batfish, both with and without abstraction. Batfish first simulates the control plane to produce the data plane and then uses NoD [78] to compute all possible packets that can traverse between source and destination nodes. With Bonsai, it takes 77 seconds to complete the query. Without it, the query did not complete and gave an out-of-memory error after running for over an hour.

4.9 Summary

In this chapter, we introduced a notion of control plane equivalence – given two SRPs and abstraction functions mapping between their topologies and attributes, every stable solution in either network has a corresponding stable solution in the other network. By finding a control-plane-equivalent abstract SRP that is smaller than the original SRP, we can scale up verification tools such as Minesweeper by running them on the smaller SRP rather than the larger counterpart. However, testing directly for control plane equivalence is as hard as the network verification problem. Instead, we identified a subset of abstractions that satisfy a collection of conditions, called effective abstractions, which imply control plane equivalence. The conditions for effective abstractions can be checked locally at each router, leading to an efficient and automated algorithm for extracting an abstract network that preserves control plane behavior. Evaluating on real and synthetic networks showed that the approach can be highly effective in compressing a network and scaling verification tools when symmetries exist in the network.

Chapter 5

Control Plane Synthesis

The previous two chapters have shown how network verification can be performed in a highly general and scalable manner. While verification can be effective for finding defects in a network after the configurations have been written, it provides no insight into how to write the configurations in the first place, or how to evolve the configurations when the network changes. Unfortunately, this means that writing configurations remains a daunting task. For example, the two real networks studied in Section 4.8 each consisted of around 600,000 lines of manually written configurations.

One of the main reasons why authoring configurations is challenging (and consequentially why misconfigurations occur frequently) is because there is a large semantic gap between the intended high-level policies that users want to enforce and the low-level mechanism through which they must express this intent. In particular, many policies involve network-wide properties—prefer a certain neighbor, ensure devices are reachable, use a particular path only if another fails—but configurations describe the behavior of individual devices. Operators must manually decompose network-wide policy into device behaviors, such that policy-compliant behavior results from the distributed interactions of these devices. Policy-compliance must be ensured, again, for all data planes that can emerge from the control plane (*e.g.*, when failures occur).

To simplify the process of configuration, in this chapter we study the application of *synthesis* to network configuration. Network synthesis asks the question: "given a high-level specification

of the desired network routing behavior, can we automatically generate configurations that are correct-by-construction?" As we will see, the answer to this question turns out to be yes. In particular, this chapter describes the design and implementation of a new network routing language and synthesis tool called Propane.

5.1 Related work

Using high-level language abstractions to help program the network routing behavior is not an entirely new idea. We briefly summarize two prior threads of work that are related to network synthesis.

Network automation: Network automation attempts to simplify configuration by automating repeated configuration tasks. For example, to reduce configuration errors, operators are increasingly adopting an approach in which common tasks are captured as parameterized templates [61, 97] to ensure certain kinds of consistency across similar devices. In addition, configuration languages such as RPSL [4], Yang [20], and Netconf [39] allow operators to express routing policy in a vendor-neutral way. While templates help ensure certain kinds of consistency across devices, they do not provide fundamentally different abstractions from existing configuration languages and thus, they still require operators to manually decompose policies into device behaviors. Additionally, they provide no guarantee that these low-level configurations satisfy any kind of high-level intent.

Another system, ConfigAssure [81, 82], is designed to help users define and debug low-level router configurations. Inputs to ConfigAssure include a *configuration database*, which contains a collection of tuples over constants and configuration variables, and a *requirement*, which is a set of constraints. The authors use a combination of logic programming and SAT solving to find concrete values for configuration variables. ConfigAssure handles configuration for a wide range of protocols and many different concerns. However, in ConfigAssure, the level of abstraction

remains relatively low as it does not offer higher-level, network-wide abstractions customized for routing and can not always ensure end-to-end correctness (*e.g.*, when failures occur).

Software-defined networks: Software-defined networking (SDN) allows users to program the network data plane directly via a set of open APIs to the device's forwarding tables. Its abstractions are, in part, the research community's response to the difficulty of maintaining policy compliance through distributed device interactions [24]. Instead of organizing networks around a distributed collection of devices that compute forwarding tables through mutual interactions, the devices are told how to forward packets by a centralized controller. The controller is responsible for ensuring that the paths taken are compliant with operator specifications.

The centralized control planes of SDN, however, are not a panacea. First, while many SDN programming systems [45] provide effective *intra*-domain routing abstractions, letting users specify paths within their network, they fail to provide a coherent means to specify *inter*-domain routes. Second, centralized control planes require careful design and engineering to be robust to failures—one must ensure that all devices can communicate with the controller at all times, even under arbitrary failure combinations. Even ignoring failures, it is necessary for the control system to scale to meet the demands of large or geographically-distributed networks, and to react quickly to environmental changes. For this challenge, researchers are exploring multi-controller systems with interacting controllers, thus bringing back distributed control planes [16, 80] and their current programming difficulties. In contrast, we will focus on directly synthesizing a distributed collection of configurations that run, without modification, on legacy hardware.

5.2 Overview

In this chapter, we have two central goals:

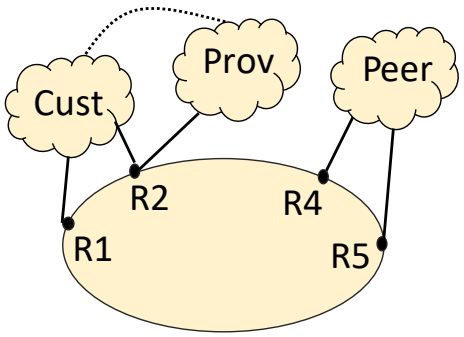
1. Design a new, high-level language with natural abstractions for expressing intra-domain routing, inter-domain routing and routing alternatives in case of failures.

2. Define algorithms for compiling these specifications into configurations for devices running standard distributed control plane algorithms, while ensuring correct behavior independent of the number of faults.

To achieve the first goal, we borrow the idea of using regular expressions to specify network paths from high-level SDN languages such as FatTire [86], Merlin [94], and NetKAT [5]. However, our design also contains several key departures from existing languages. The most important one is semantic: the paths specified can extend from outside the operator's network to inside the network, across several devices internally, and then out again. This design choice allows users to specify preferences about both external and internal routes in the exact same way. In addition, we augment the algebra of regular expressions to support a notion of *preferences* and provide a semantics in terms of sets of ranked paths. The preferences indicate fail-over behaviors: among all specified paths that are still available, the system guarantees that the distributed implementation will always use the highest-ranked ones. Although we target a distributed implementation, the language is more general and could potentially be used in an SDN context.

To achieve the second goal, we develop program analysis and compilation algorithms that translate the regular policies to a graph-based intermediate representation and from there to per-device BGP configurations, which include the various filters and preferences that govern BGP behavior. We target BGP for pragmatic reasons: it is a highly flexible routing protocol, it is an industry standard, and many networks use it internally as well as externally. Despite the advent of SDN, many networks will continue to use BGP for the foreseeable future due to existing infrastructure investments, the difficulty of transitioning to SDN, and the scalability and fault-tolerance advantages of a distributed control plane.

Policy-compliance: The BGP configurations produced by our compiler are guaranteed to be policy-compliant in the face of *arbitrary* failures. This guarantee does not mean that the implementation is always able to send traffic to its ultimate destination (*e.g.*, in the case of a network partition), but rather that it always respects the centralized policy, which may include dropping



Policy

P1. Prefer Cust > Peer > Prov
P2. Disallow traffic between Prov and Peer
P3. For Cust, prefer R1 > R2
P4. Cust must be on path for its prefixes
P5. Cust must not be a transit to Prov

Figure 5.1: Creating router-level policies is difficult.

traffic when there is no route. In this way, we provide network operators with a strong guarantee that is otherwise impossible to achieve today. However, some policies simply cannot be implemented correctly in BGP in the presence of arbitrary failures. We develop new algorithms to detect such policies and report our findings to the operators, so they may fix the policy specification at compile time rather than experience undesirable behavior after the configurations are deployed.

5.3 Example Network Policies

When generating BGP configurations, whether manually or aided by templates, the operators face the challenge of decomposing network-wide policies into correct device-level policies. This decomposition is not always straightforward and ensuring policy-compliance is tricky, especially in the face of failures. In this section, we illustrate this difficulty using two examples based on policies that we have seen in practice. The next section shows how Propane allows operators to express these policies naturally.

Example 1: the backbone: Consider the backbone network in Figure 5.1. The network has three neighbors, a customer `Cust`, a peer `Peer`, and a provider `Prov`. The policy of this network is shown on the right, and can be summarized as follows:

P1 The policy prefers that traffic leave the network through neighbors in a certain order

P2 The policy does not want the network to act as a transit between `Peer` and `Prov`

P3 The policy prefers that the network exchange traffic with `Cust` over `R1` rather than `R2` because `R1` is cheaper

P4 To guard against an AS "hijacking" prefixes owned by `Cust`, the network should only send `Cust` traffic to a neighbor if `Cust` is on the AS path

P5 To guard against `Cust` accidentally becoming a transit for `Prov`, the network should not use `Cust` for traffic that will later traverse `Prov`

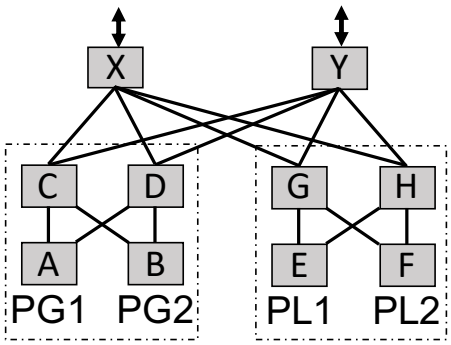
To implement policy P1, the operators must compute and assign local preferences such that preferences at `Cust`-facing interfaces $>$ `Peer`-facing interfaces $>$ `Prov`-facing interfaces. At the same time, to satisfy P3, the preference at `R2`'s `Cust`-facing interface should be lower than that at `R1`. Implementing P3 will also require MEDs to be appropriately configured on `R1` and `R2`. To implement P2, the operators can assign communities that indicate where a certain routing announcement entered the network. Then, `R4` must be configured to not announce to `Peer` routes that have communities that correspond to the `R2-Prov` link but to announce routes with communities for the `R2-Cust` and `R1-Cust` links. A similar type of policy must be configured for `R2` as well. Finally, to implement P4 and P5, the operators will have to compute and configure appropriate prefix- and AS-path-based import and export filters at each router. To summarize, implementing this policy requires the following low-level configuration:

P1 Set local-preference higher on import at `R1–Cust` and `R2–Cust` than at `R4–Peer` and `R5–Peer`, which is higher than at `R2–Prov`

P2 Attach a community value on import along interfaces `R4–Peer`, `R5–Cust`, and `R2–Prov`, and drop messages with this community on export for the same interfaces.

P3 Set the Multi-Exit Discriminator lower on export at `R1–Cust` than at `R2–Cust`

P4 On import, drop routes along `R1–Cust` and `R2–Cust` that do not match the regular expression (`Cust` $\cdot .^*$)



Policy

P6. Left cluster has global services with PG* prefixes, which should be announced externally as an aggregate PG

P7. Right cluster has local services with PL* prefixes, which should not be announced externally

Figure 5.2: Policy-compliance under failures is difficult.

P5 On import, drop any route that matches the regular expression $(.^* \cdot \texttt{Cust} \cdot .^* \cdot \texttt{Prov})$

Clearly, it is difficult to correctly configure even this small example network manually; correctly configuring real, larger networks can quickly become a nightmare. Such networks have hundreds of neighbors spanning multiple commercial-relationship classes, differing numbers of links to each neighbor, along with several neighbor- or prefix-based exceptions to the default behavior. A large AS with many peers in different geographic locations may be faced with complex challenges such as keeping traffic within national boundaries. Templates help to an extent by keeping preference and community values consistent across routers, but operators must still do much of the conceptually difficult work manually.

Example 2: the datacenter: While configuring policies for a fully functional network is difficult, ensuring policy compliance in the face of failures can be almost impossible. Consider the datacenter network in Figure 5.2 with routers organized as a fat tree and running BGP with private AS numbers [71]. The network has two clusters, one with services that should be reachable globally and one with services that should be accessible only internally. This policy is enabled by using non-overlapping address space in the two clusters and ensuring that only the address space for the global services is announced externally. Further, to reduce the number of prefixes that are announced externally, the global space is aggregated into a less-specific prefix `PG`.

The operator may implement the policy by having X and Y: not export externally what they hear from G and H, routers that belong to the local services cluster (P6); and export externally what

they hear from routers C and D and aggregate to `PG` if an announcement is a subset of `PG` (P7). This implementation is appealing because X and Y do not need to be made aware of which prefixes are global versus local and IP address assignment can occur independently, *e.g.*, local services can be assigned new prefixes without updating those routers' configurations.

However, this implementation has incorrect behavior in the face of failures. Suppose links X–G and X–H fail. Then, X will hear announcements for `PL*` from C and D, having traversed from G and H to Y to C and D. Per its policy implementation, X will start "leaking" these prefixes externally. Depending on the rationale for local services, this leak could impact security (*e.g.*, if the services are sensitive) or availability (*e.g.*, if the `PL*` prefixes are reused for other services outside of the datacenter). This problem does not manifest without failures because then X has and prefers paths to `PL*` through G and H since they are shorter. A similar problem will occur if links Y–G and Y–H fail. Link failures in datacenters are frequent and it is not uncommon to have many failed links at a given time [51].

To avoid this problem, the operator may disallow "valley" paths, *i.e.*, those that go up, down, and back up again. This guard can be implemented by X and Y rejecting paths through each other. But that creates a different problem in the face of failures—an aggregation-induced black hole [73]. If links D–A and X–C fail, X will hear an announcement for `PG2` from D and will thus announce `PG` externally. This announcement can bring in traffic for `PG1` to X as well, but because valleys are disallowed, X does not have a valid route for `PG1` and will drop all traffic this destination despite the fact that a valid path exists through Y.

Thus, we see that devising a configuration that ensures policy compliance in the face of failures is complex and error-prone. Propane lets operators implement their high-level policy specification in a way that guarantees compliance under all failures if possible—otherwise, it generates a compile-time error. For aggregation, it also provides a lower bound to operators on the number of failures under which aggregation will not result in black holes.

5.4 A Routing Language

Policies for (distributed) control planes differ from data-plane policies in a few important ways. First, they must account for all failures at compile time; there is no controller at runtime, so the routers must be configured in advance to handle failures in a compliant manner. In Propane, we enable such specifications through *path preferences*, with the semantics that a less-preferred path is taken only when a higher-preference path is unavailable in the network. Second, paths in a control-plane policy may be under-specified (*e.g.*, "prefer customer" does not indicate a concrete path). The Propane compiler treats such under-specifications as constraints on the set of allowed paths and automatically computes valid sets based on the topology.

This section introduces the Propane language using the examples from the previous section and provides the complete syntax of the language. The next section describes our strategy for compiling it to BGP.

Example 1: The backbone: Propane allows operators to configure the network with the abstraction that they have centralized control over routing. Specifically, the operator simply provides a set of high-level constraints that describe the paths traffic should—or should not—take and their relative preferences. Propane specifications are written modularly via a series of declarations. For example, to begin specification of the backbone network from the previous section, we first express the idea that we prefer that traffic leave the network through `R1` over `R2` (to `Cust`) over `Peer` over `Prov` (policy P1 and P3 from Figure 5.1):

```
define Prefs = exit(R1 >> R2 >> Peer >> Prov)
```

This statement defines a set of *ranked paths*, which includes all paths (and only those paths) for which traffic exits our network through either router `R1`, router `R2`, `Peer`, or `Prov`. The paths that exit through `R1` are preferred (>>) to those that exit through `R2`, which are preferred to those that leave through `Peer` and then `Prov`. As we describe in the next section, the **`exit`** expression, as well as other path expressions used later in this section, is simply a shorthand for a particular regular expression over paths that is expressible in our policy language. The preference operator (>>)

is flexible and can be used between constraints as well as among individual routers. For example, the above constraint could have been written equivalently as **exit**(R1) >>...>> **exit**(Prov)

To associate ranked paths with one or more prefixes, we define a Propane *policy*. Within a policy, statements with the form t => p associate the prefixes defined by the predicate t with the set of ranked paths defined by the path expression p. In general, prefix predicates can be defined by arbitrary boolean combinations (and, or, not) of concrete prefixes. Here, we assume we have already defined the predicate PCust for the customer prefixes. In the following code, ranked paths are associated with customer prefixes, and all other prefixes (true). Policy statements are processed in order with earlier policy statements taking precedence over later policy statements. Hence, when the predicate true follows the statement involving PCust, it is interpreted as true & !PCust.

```
define Routing =
    {PCust => Prefs & end(Cust)
     true  => Prefs }
```

Line 2 of this policy restricts traffic destined to known customer prefixes (PCust) to only follow paths that end at the customer. In addition, it enforces the network-wide preference that traffic leaves through R1 over R2 over Peer over Prov. Line 3 applies to any other traffic not matching PCust and allows the traffic to leave through any direct neighbor with the usual preference of R1 over R2 over Peer over Prov. To summarize our progress, the Routing policy implements P1, P3, and P4 from Figure 5.1.

Since, routing allows transit traffic by default (*e.g.*, traffic entering from Peer and leaving through Prov), we separately define a policy to enforce P2 and P5 from Figure 5.1, using conjunction (&), disjunction (|) and negation (!) of constraints. First, we create reusable abstractions for describing traffic that transits our network. In Propane, this is done by creating a new parameterized definition.

```
define transit(X,Y) = enter(X|Y) & exit(X|Y)
define cust-transit(X,Y) = later(X) & later(Y)
```

Here we define transit traffic between groups of neighbors X and Y as traffic that enters the network through some neighbor in X or Y and then also leaves the network through some neighbor

```
define Prefs = exit(R1 >> R2 >> Peer >> Prov)

define Routing =
    {PCust => Prefs & end(Cust)
      true  => Prefs }

define transit(X,Y) = enter(X|Y) & exit(X|Y)
define cust-transit(X,Y) = later(X) & later(Y)

define NoTrans =
  {true => !transit(Peer,Prov) &
           !cust-transit(Cust,Prov)}

Routing & NoTrans
```

Figure 5.3: Complete Propane policy for the backbone network.

in either X or Y. Similarly, we define customer transit for customer X and provider Y as traffic that later goes through both X and Y after leaving our network. Using these two new abstractions, we can now implement policies P2 and P5 with the following constraint.

```
define NoTrans =
  {true => !transit(Peer,Prov) &
           !cust-transit(Cust,Prov)}
```

The `NoTrans` constraint requires that all traffic not follow a path that transits our network between `Peer` and `Prov`. Additionally, it prevents traffic from ever following paths that leave our network and later go through both `Prov` and `Cust`. To implement both `Routing` and `NoTrans` simultaneously, we simply conjoin them: `Routing & NoTrans`.

Collectively, the constraints above capture the entire policy, which is shown in Figure 5.3. From them, our compiler will generate per-device BGP import and export filters, local preferences, MED attributes, and community tags to ensure that the policy is implemented correctly under all failures.

Example 2: The datacenter: Our datacenter example network has three main concerns: (1) traffic for the prefix allocated to each top-of-rack router must be able to reach that router, (2) local

services must not leak outside the datacenter, and (3) aggregation must be performed on global prefixes to reduce churn in the network.

Propane allows modular specification of each of these constraints. The first constraint is about prefix ownership—we want traffic only for certain prefixes to end up at a particular location. The following definition captures this intent.

```
define Ownership =
    {PG1 => end(A)
     PG2 => end(B)
     PL1 => end(E)
     PL2 => end(F)
     true => end(out)}
```

This definition says that traffic for prefix `PG1` is allowed to follow only paths that end at router `A`; traffic for `PG2`, but not `PG1`, must end at router `B`; and so on. Any traffic destined for a prefix that is not a part of the datacenter should be allowed to leave the datacenter and end at some external location, which is otherwise unconstrained. The special keyword **`out`** matches any location outside the datacenter network, while the keyword **`in`** will match any location inside the network.

For the second constraint, we define another policy:

```
define Locality = {PL1 | PL2 => only(in)}
```

This definition says that traffic for local prefixes only follows paths that are internal to the network at each hop. This constraint guarantees that the services remain accessible only to locations inside the datacenter. While not part of the original datacenter policy, it is also simple and natural to reuse the definition of transit traffic from the backbone network in Propane. For instance, we could prevent the data center from serving transit between external peers by writing:

```
define NoTrans = {true => !transit(out, out)}
```

As in the backbone example, we can logically conjoin these constraints to specify the network-wide policy. However, in addition to constraints on the shape of paths, Propane allows the operator to specify constraints on the BGP control plane itself. For instance, a constraint on aggregation is included to ensure that aggregation for global prefixes is performed from locations inside (**`in`**) the

```
define Ownership =
    {PG1 => end(A)
    PG2 => end(B)
    PL1 => end(E)
    PL2 => end(F)
    true => end(out)}

define Locality =
    {PL1 | PL2 => only(in)}

define transit(X,Y) = enter(X|Y) & exit(X|Y)
define NoTrans =
    {true => !transit(out,out)}

Ownership & Locality &
NoTrans & agg(PG, in -> out)
```

Figure 5.4: Complete Propane policy for the data center network.

network to locations outside (**out**). In this case, `PG1` and `PG2` will use the aggregate `PG` (which we assume is defined earlier) when advertised outside the datacenter.

```
Ownership & Locality & NoTrans & agg(PG, in -> out)
```

The complete datacenter policy is shown in Figure 5.4. Once Propane compiles the policy, it is guaranteed to remain compliant under all possible failure scenarios, modulo any aggregation-induced black holes. In the presence of aggregation, the Propane compiler will also efficiently find a lower bound on the number of failures required to create an aggregation-induced black hole.

5.4.1 Regular IR (RIR)

The Propane syntax used in the examples is just a thin layer atop a regular-expression-based core language (RIR) for describing preference-based path constraints. Figure 5.5 shows the RIR syntax. A policy has one or more constraints. The first kind of constraint is a test on the type of route and a corresponding set of preferred regular paths. Regular paths are regular expressions where the base characters are abstract locations representing either a router or an external AS. Special **in** and **out**

$$
\begin{array}{rcll}
pol & ::= & p_1, \ldots, p_n & \textit{policies} \\
p & ::= & t \texttt{ => } r_1 \texttt{>>} \ldots \texttt{>>} r_m \mid cc & \textit{constraints} \\
x & ::= & d.d.d.d/d & \textit{prefix} \\
t & ::= & \texttt{true} & \textit{true} \\
 & \mid & \texttt{!}\, t & \textit{negation} \\
 & \mid & t_1 \texttt{|}\, t_2 & \textit{disjunction} \\
 & \mid & t_1 \texttt{\&} t_2 & \textit{conjunction} \\
 & \mid & \textit{prefix} = x & \textit{prefix test} \\
 & \mid & \textit{comm} = d & \textit{community test} \\
r & ::= & l & \textit{location} \\
 & \mid & \emptyset & \textit{empty set} \\
 & \mid & \texttt{in} & \textit{internal loc} \\
 & \mid & \texttt{out} & \textit{external loc} \\
 & \mid & r_1 \cup r_2 & \textit{union} \\
 & \mid & r_1 \cap r_2 & \textit{intersection} \\
 & \mid & r_1 \cdot r_2 & \textit{concatenation} \\
 & \mid & \texttt{!}\, r & \textit{path negation} \\
 & \mid & r^* & \textit{iteration} \\
ln & ::= & r_1 \rightarrow r_2 & \textit{links} \\
cc & ::= & agg(x, ln) \mid tag(d, t, ln) & \textit{control constraints}
\end{array}
$$

Figure 5.5: Regular Intermediate Representation syntax.

symbols refer to any internal or external location respectively. In addition, Σ refers to any location. We also use the standard regular expression abbreviation r^+ for $r \cdot r^*$, a sequence of one or more occurrences of r. Predicates (t) consist of logical boolean connectives (and, or, not) as well as tests that match a particular prefix (or group of prefixes) and tests for route advertisements with a particular community value attached (*i.e.*, an integer value associated with a path). As an example, the RIR constraint

$$(\text{prefix} = 74.3.28.0/24) \texttt{ => } (\text{as}200 \cdot \texttt{in}^+) \texttt{ >> } (\text{as}100 \cdot \texttt{in}^+)$$

describes a more-preferred set of paths for traffic announced by a prefix no less specific than `74.125.28.0/24`, which starts at AS 200, before entering and staying inside the user's network to get to the destination, and a less-preferred set of paths that start at AS 100 and are otherwise the same. The plus operator $\texttt{in}^+$ ensures there must be at least one hop internally.

Propane also supports constraints on the control-plane behavior of BGP. For example, prefix aggregation is an important optimization to reduce routing table size. A constraint of the form $\mathrm{agg}(x, ln)$ tells the compiler to perform aggregation for prefix x across all links described by ln. It is also often useful to be able to add community tags to exported routes in BGP (*e.g.*, to communicate non-standard information to peers). A constraint of the form $\mathrm{tag}(d, t, ln)$ adds community tag d for any prefixes matching t across links ln. Aggregation, for example, from internal to external locations, is specified using the same regular syntax as before:

$$\texttt{agg}(128.17.0.0/16, \texttt{in->out})$$

where the expression `in->out` refers to control messages flowing from any internal to any external location. We list only the route aggregation and community tagging constraints in Figure 5.5, but Propane also supports other constraints such as limiting the maximum number of routes allowed between ASes, or enabling BGP multipath.

5.4.2 Semantics

We give a semantics to RIR programs using sets of ranked paths. Each path constraint $r_1 >> \ldots >> r_j$ denotes a set of ranked network paths. A network path is a topologically valid string of abstract locations $l_1 l_2 \ldots l_k$ that is loop-free. We use the notation $|p|$ to denote the length of the path p. A regular expression r matches path p, if $p \in \mathcal{L}(r)$, that is, the path is in the language of the regular expression. We write $\mathrm{suffix}(p, q)$ to mean that q is a subpath of p, *i.e.*, if $p = l_1, \ldots, l_n$ then $q = l_j, \ldots, l_n$ where $j \geq 1$. The semantics for a policy in Propane, then can be defined to produce a set of ranked paths (pairs of a path and a rank, where a rank is itself a pair of natural numbers) as follows:

$$\begin{aligned}
[\![r]\!]_{(T,i)} &= \{(q, (i, |q|)) \mid \mathrm{suffix}(p, q), p \in \mathcal{L}(r), p \text{ is a network path in } T\} \\
[\![r_1 >> \ldots >> r_j]\!]_T &= [\![r_1]\!]_{(T,1)} \cup \ldots \cup [\![r_j]\!]_{(T,j)}
\end{aligned}$$

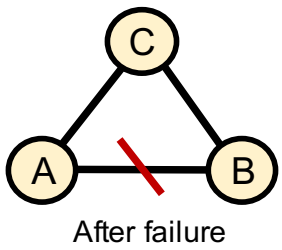


Figure 5.6: Propane semantics example with a failure.

The semantics takes a network topology T and produces a set of ranked paths. The rank of a path is a lexicographic order of two values: (1) the priority of the regular expression matched, and (2) as a tie breaker, the path length. Lower ranks indicate *more* preferred paths. The set of ranked paths depends on which paths are valid in the topology (*i.e.*, a network path), and thus when failures occur, the most preferred routes change. The inclusion of all suffixes of a path in the semantics ensures that if path p is allowed by a policy, then each sub-path along path p is also allowed (otherwise most policies would be trivially unimplementable). For any source s and destination d, Propane will send traffic along the best (lowest ranked) available path from s to d. The Propane compiler must ensure that generated configurations for a policy always achieve the most preferred paths possible given the failures in the topology, using only distributed mechanisms.

Example: Consider the policy: `true => (A · B · C) >> (in*· C)` applied to the network in Figure 5.6. Here, **`in`** refers to any node in the network (all nodes are internal). If there are no failures, then the semantics of this policy produces the following set (shown organized by path rank):

$$\{(\texttt{ABC}, (1,3)), (\texttt{BC}, (1,2)), (\texttt{C}, (1,1)),$$
$$(\texttt{ABC}, (2,3)), (\texttt{BC}, (2,2)), (\texttt{C}, (2,1)), (\texttt{AC}, (2,2)), (\texttt{BAC}, (2,3))\}$$

The first preference regular expression (`A · B · C`) produces the set of paths with rank 1:

$$\{(\texttt{ABC}, (1,3)), (\texttt{BC}, (1,2)), (\texttt{C}, (1,1))$$

The latter two paths in this set are included because they are suffixes of the matching path ABC. The second preference regular expression (**in**$^*\cdot$ C) produces the remaining paths, which have rank 2. For each source, destination node pair, we chose the lowest ranked path between these nodes according to the semantics. In this case the resulting paths that must be used for forwarding are: {ABC, BC, C} since each such path has a rank of 1. Now, consider the case where the AB link has failed. In this case, we get the set of ranked paths:

$$\{(\texttt{C}, (1,1)), (\texttt{C}, (2,1)), (\texttt{BC}, (1,2)), (\texttt{BC}, (2,2)), (\texttt{AC}, (2,2))\}$$

Once again, picking the best ranked paths for each source/destination pair, we get the set of paths: {AC, BC, C}. This time, the most preferred path of ABC is no longer available, but the next most preferred path for A (AC) is available.

It is the job of the Propane compiler to generate BGP policies running on each device that ensure the BGP protocol will always correctly find the best possible paths for each source/destination pair, even when such failures occur.

5.4.3 Limitations

While the combination of regular expressions and preferences allows operators to describe a wide variety of policies, there are many Propane policies that are either inexpressible, or unimplementable in BGP. One such class of policies is those policies that control traffic outside of the user's network. For instance, suppose an operator writes a policy such as (**out**$^*\cdot$ Cust $\cdot$ Peer $\cdot$ **out***), which talks only about networks not under the operator's control. Clearly, without access to the customer and peer network configurations, such a policy can not be implemented, as there is no way to influence how the customer and peer route their own traffic through each other. In general, one can only write Propane policies that go through the network currently being configured. The policy also can not go through the current network more than once (*e.g.*, in from a peer, out to another peer, then back in). Another limitation is that, while it is possible to prefer or reject

routes learned upstream (*i.e.*, routes learned from neighbors), it is generally not possible to control where a route travels downstream (*i.e.*, after the route has left the network). A special "no-export" community can be attacked to routes to ensure the route is not exported beyond the immediate neighbor, but this is the extent of the control provided to an operator.

These constraints taken together mean that, for each regex r in a Propane policy, r must satisfy one of two conditions. The first is that $r \subseteq \texttt{out}^* \cdot \texttt{in}^+ \cdot (\epsilon + \texttt{out})$. In other words, the regular expression only references paths that go through at least one internal node, and either do not leave the network, or only extend a single hop outside the network. The other possibility is that the regular expression can be viewed as the concatenation of two regular expressions where $r = s \cdot t$, and $s \subseteq \texttt{out}^* \cdot \texttt{in}^+$ represents the valid paths that go through at least one hop in the network, and $t \supseteq \texttt{out}^*$ meaning that t is too general to restrict the policy to any specific external nodes.

Finally, as we will see later, there are some combinations of policy preferences that can not be implemented correctly for all failures for the BGP protocol.

One might also think that there are policies that can not be implemented because they are inconsistent (*e.g.*, $(\texttt{drop} \ \& \ \texttt{any})$). However the semantics of the language resolves any such ambiguities. In this example, the intersection of the set for **any** and that for **drop** is the empty set indicating that traffic should be dropped.

5.5 Compilation

The language defined in the previous section is general enough to describe a wide variety of different routing policies such as those from Section 5.3. However, several problems remain. First, we must be able to decompose policies written in this language into purely distributed mechanisms that can be implemented using only local processing on devices. And second, we must implement such policies working within the limitations of the BGP routing protocol. The rest of this section describes, step-by-step, how to turn a Propane policy into a collection of BGP configurations.

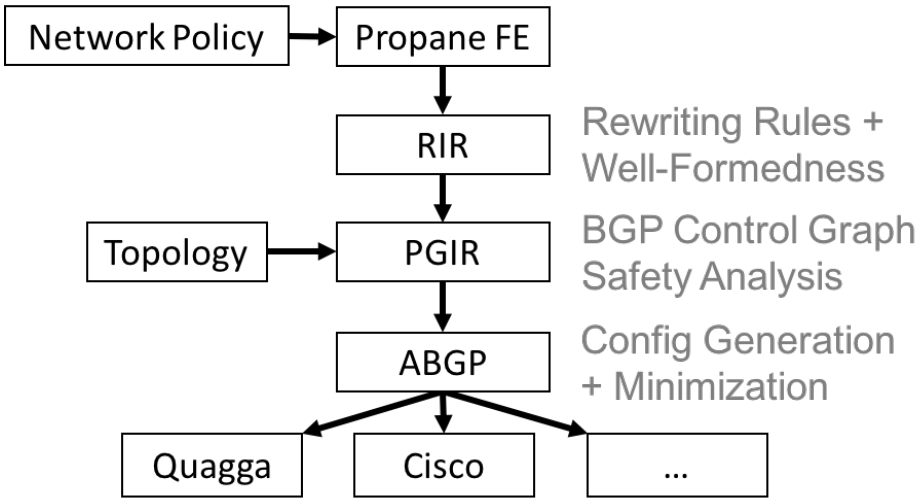


Figure 5.7: Compilation pipeline stages for Propane.

To handle these challenges, we decompose compilation into multiple stages, shown in Figure 5.7, and develop efficient algorithms for the translation between stages. The first stage of the pipeline involves simple rewriting rules and substitutions from the front-end language (FE) to the core Regular Intermediate Representation (RIR). Policies in RIR are checked for well-formedness (*e.g.*, never constraining traffic that does not go through the user's network), before being combined with the topology to obtain the Product Graph Intermediate Representation (PGIR). The PGIR is a data representation that compactly captures the flow of BGP announcements subject to the policy and topology restrictions. We develop efficient algorithms that operate over the PGIR to ensure policy compliance under failures, avoid BGP instability, and prevent aggregation-induced black holes. Once the compiler determines safety, it translates the PGIR to an abstract BGP (mBGP) representation. mBGP can then be translated into various vendor-specific device configurations as needed. The Propane compiler currently generates Quagga [85] router configurations from mBGP.

5.5.1 From FE to RIR

The first stage in Propane compilation reduces the front-end (FE) language used in the examples to the simpler RIR from Figure 5.5. The main differences between the FE and RIR are: (1) FE allows the programmer to specify constraints using a series of (modular) definitions, and combine them

$$
\begin{array}{rcl}
\textbf{any} & = & \textbf{out}^* \cdot \textbf{in}^+ \cdot \textbf{out}^* \\
\textbf{drop} & = & \emptyset \\
\textbf{internal} & = & \textbf{in}^+ \\
\textbf{only}(X) & = & \textbf{any} \cap X^* \\
\textbf{never}(X) & = & \textbf{any} \cap (!X)^* \\
\textbf{through}(X) & = & \textbf{out}^* \cdot \textbf{in}^* \cdot X \cdot \textbf{in}^* \cdot \textbf{out}^* \\
\textbf{later}(X) & = & \textbf{out}^* \cdot (X \cap \textbf{out}) \cdot \textbf{out}^* \cdot \textbf{in}^+ \cdot \textbf{out}^* \\
\textbf{before}(X) & = & \textbf{out}^* \cdot \textbf{in}^+ \cdot \textbf{out}^* \cdot (X \cap \textbf{out}) \cdot \textbf{out}^* \\
\textbf{end}(X) & = & \textbf{any} \cap (\Sigma^* \cdot X) \\
\textbf{start}(X) & = & \textbf{any} \cap (X \cdot \Sigma^*) \\
\textbf{exit}(X) & = & (\textbf{out}^* \cdot \textbf{in}^* \cdot (X \cap \textbf{in}) \cdot \textbf{out} \cdot \textbf{out}^*) \cup (\textbf{out}^* \cdot \textbf{in}^+ \cdot (X \cap \textbf{out}) \cdot \textbf{out}^*) \\
\textbf{enter}(X) & = & (\textbf{out}^* \cdot \textbf{out} \cdot (X \cap \textbf{in}) \cdot \textbf{in}^* \cdot \textbf{out}^*) \cup (\textbf{out}^* \cdot (X \cap \textbf{out}) \cdot \textbf{in}^+ \cdot \textbf{out}^*) \\
\textbf{link}(X, Y) & = & \textbf{any} \cap (\Sigma^* \cdot X \cdot Y \cdot \Sigma^*) \\
\textbf{path}(\vec{X}) & = & \textbf{any} \cap (\Sigma^* \cdot X_1 \ldots X_n \cdot \Sigma^*) \\
\textbf{novalley}(\vec{X}) & = & \textbf{any} \cap !\textbf{path}(X_2, X_1, X_2) \cap \cdots \cap !\textbf{path}(X_n, X_{n-1}, X_n)
\end{array}
$$

Figure 5.8: Propane language expansions.

later, (2) FE provides high-level names that abstract sets of routes and groups of prefixes/neighbors, and (3) FE allows the preference operator to be used more flexibly.

Merging constraints: The translation from FE to RIR is based on a set of rewriting rules. The first step is to check for well-formedness according to the conditions specified in Section 5.4.3. The next step merges separate constraints. It takes the cross product of per-prefix constraints, where logical conjunction (r_1 `&` r_2) is replaced by intersection on regular constraints ($r_1 \cap r_2$), logical disjunction is replaced by union, and logical negation (`!`r) is replaced by path negation (**any** $\cap$ `!`(r)). The additional constraint **any** ensures the routes are well-formed by restricting the paths to only those that go through the user's network. For example, in the datacenter FE configuration from Section 5.3, combining the `Locality` and `Ownership` policies results in the following RIR:

```
PG1 => end(A)
PG2 => end(B)
PL1 => only(in) ∩ end(E)
PL2 => only(in) ∩ end(F)
true => exit(out)
```

Lifting preferences: Since preferences can only occur at the outermost level for an RIR expression, the next step is to "lift" occurrences of the preference operator in each regular expression. For example, the regular expression $r \cdot (s >> t) \cdot u$ is lifted to $(r \cdot s \cdot u)>>(r \cdot t \cdot u)$ by distributing the preference over the sequence operator. In general, we employ the following distributivity equivalences:

$$r \odot (s_1>> \ldots >>s_n) = (r \odot s_1)>> \ldots >>(r \odot s_n)$$

$$(s_1>> \ldots >>s_n) \odot r = (s_1 \odot r)>> \ldots >>(s_n \odot r)$$

where $\odot$ stands for an arbitrary regular binary operator, and r is a policy with a single preference. In cases where r does not contain a single preference, such as $(s >> t) \cdot (u >> v)$, it is not clear which of the paths $s \cdot v$ or $t \cdot u$ is preferred. Propane rejects such ambiguous policies, requiring instead that operators explicitly specify which paths to prefer — for example as $(s \cdot u) >> (s \cdot v) >> (t \cdot u) >> (t \cdot v)$. Policies that contain preferences nested under a unary operator (*i.e.*, *star* or *negation*) are also rejected by Propane as invalid.

Regular constraint rewriting: The final step of compilation normalizes policies by rewriting the high-level constraints such as **`end`** into pure regular expressions. This rewriting is defined according to the equivalences in Figure 5.8. For example, the constraint **`end`**(A) will turn into **`any`** $\cap$ $(\Sigma^* \cdot \texttt{A})$. This rewriting desugars the high level constraint descriptions like "end" into actual regular expressions that define the shape of a path.

5.5.2 Product Graph IR

Product graph IR: Now that the user policy exists in a simplified form, we must consider the topology. In particular, we want a compact way to represent all the possible ways BGP route announcements can flow through the network subject to the policy and topology constraints. The PGIR is a data structure that captures exactly this information by "intersecting" each of the regular automata corresponding to the RIR path preferences with the topology. This idea of intersection regular constraints with a graph has been used in the past in the database literature [75], and more

recently in networks [94]. Paths through the PGIR correspond to real paths through the topology that also satisfy one or more of the user constraints.

Formal definition: While paths in an RIR policy describe the direction traffic flows through the network, to implement the policy with BGP we are concerned about the way control-plane information is disseminated — route announcements flowing in the opposite direction. To capture this idea, when compiling an RIR expression of the form $r_1 >> \ldots >> r_k$, for each regular expression r_i in an RIR policy we construct a deterministic finite state machine for the reversed regular expression. An automata for regular expression r_i is defined as a tuple $(\Sigma, Q_i, F_i, q_{0_i}, \sigma_i)$. The alphabet Σ consists of all abstract locations (*i.e.*, routers or ASes), Q_i is the set of states for automaton i, F_i is the set of final states, q_{0_i} is the initial state, and $\sigma_i \colon Q_i \times \Sigma \to Q_i$ is the state transition function. The topology is represented as a graph (V, E), which consists of a set of vertices V and a set of directed edges $E \colon V \times V$. The combined PGIR is a tuple (V', E', s, e, rank) with vertices $V' \colon V \times Q_1 \times \cdots \times Q_j$, edges $E' \colon V' \times V'$, a unique starting vertex s, a unique ending vertex e, and a preference function $\mathrm{rank} \colon V' \to 2^{\{1,\ldots,j\}}$, which maps nodes in the product graph to a set of path ranks. Hence, the nodes in the product graph are tuples of a concrete topology node and a state from each automata.

For a PGIR vertex $n = (l, \ldots) \in V'$, we say that n is a *shadow* of topology location l. We also write $\tilde{n} = l$ to indicate that the topology location for node n is l. When two PGIR nodes m and n shadow the same topology location (*i.e.*, $\tilde{m} = \tilde{n}$), we write $m \approx n$.

Throughout the remainder of this chapter, we will use the convention that metavariables m and n stand for PGIR nodes and l stands for a topology location. Capital letters like X refer to concrete topology locations, while capital letters with subscripts such as X_1 and X_2 refer to concrete PGIR nodes that share a topology location (*i.e.*, $\tilde{X_1} = \tilde{X_2} = X$).

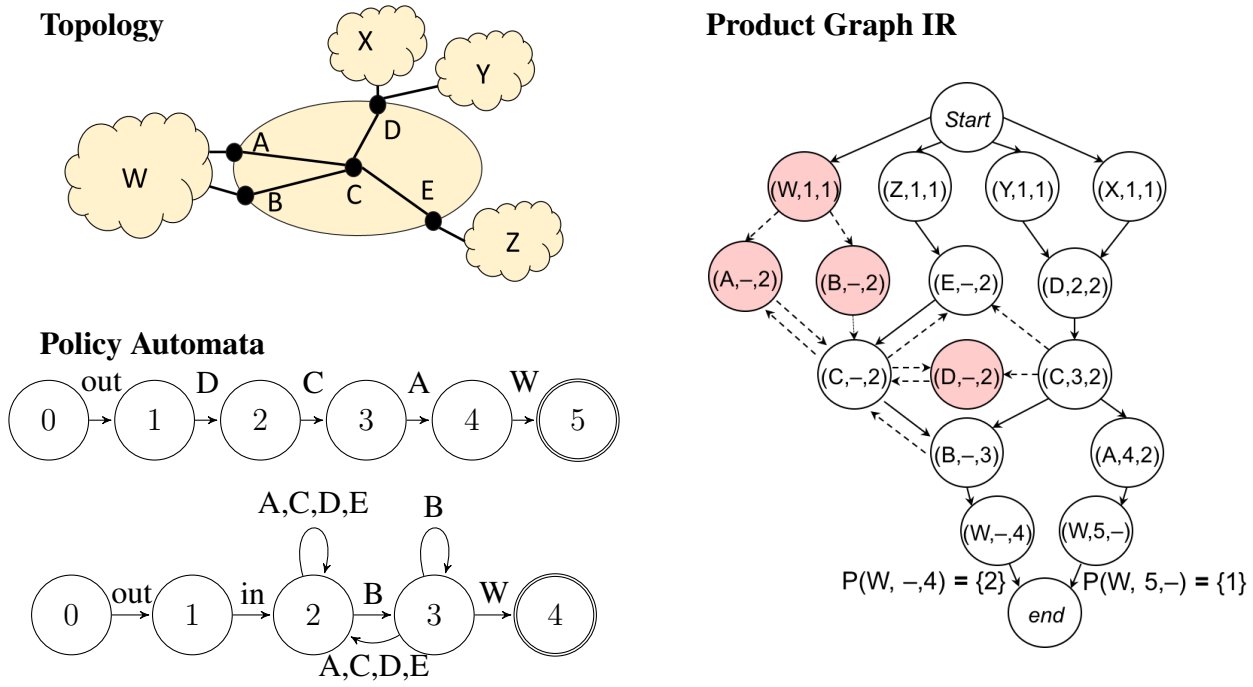


Figure 5.9: Product graph construction for policy $(\texttt{W} \cdot \texttt{A} \cdot \texttt{C} \cdot \texttt{D} \cdot \textbf{out}) >> (\texttt{W} \cdot \texttt{B} \cdot \textbf{in}^{+} \cdot \textbf{out})$.

5.5.3 From RIR To PGIR

Let a_i and b_i denote states in the regular policy automata. The PGIR is constructed by adding an edge from $m = (l_m, a_1, \ldots, a_k)$ to $n = (l_n, b_1, \ldots, b_k)$ whenever $\sigma_i(a_i, l_n) = b_i$ for each i and $(l_m, l_n) \in E$ is a valid topology link. Additionally, we add edges from the start node s to any $n = (l, a_1, \ldots, a_k)$ when $\sigma_i(q_{0_i}, l) = a_i$ for each i. The preference function rank is defined as $\mathrm{rank}(n) = \{i \mid a_i \in F_i\}$. That is, it records the path rank of each automaton that has reached a final state. Finally, there is an edge from each node in the PGIR such that $\mathrm{rank}(n) \neq \emptyset$ to the special end node e.

Intuitively, the PGIR tracks the policy states of each automaton as route announcements move between locations. Consider the topology in Figure 5.9. Suppose we want a primary route from neighbor W that allows traffic to enter the network at A and utilize the C–D link before leaving the network (through X or Y). As a backup, we also want to allow traffic to enter the network from B, in which case the traffic can also utilize the C–E link before leaving the network. For simplicity,

we assume that the route ends in either X, Y, or Z. The RIR for the policy could be written as:

$$(\texttt{W} \cdot \texttt{A} \cdot \texttt{C} \cdot \texttt{D} \cdot \textbf{out}) >> (\texttt{W} \cdot \texttt{B} \cdot \textbf{in}^{+} \cdot \textbf{out})$$

Figure 5.9 shows the policy automata for each regular expression preference. Since we are interested in the flow of control messages, the automata match backwards. The figure also shows the PGIR after intersecting the topology and policy automata. Every path in the PGIR corresponds to a concrete path in the topology. In particular, every path through the PGIR that ends at a node n such that the preference function $\text{rank}(n) = \{i_1, \ldots, i_k\}$ is non-empty, is a valid topological path that satisfies the policy constraints and results in a particular path with preferences i_1 through i_k. For example, the path $\texttt{X} \cdot \texttt{D} \cdot \texttt{C} \cdot \texttt{A} \cdot \texttt{W}$ is a valid path in the topology that BGP route announcements might take, which would lead to obtaining a path with the lowest (best) rank of 1. BGP control messages can start from peer X, which would match the **out** transition from both automata, leading to state 1 in the first automaton, and state 1 in the second automaton. This possibility is reflected in the product graph by the node with state (X,1,1). From here, if X were to advertise this route to D, it would result in the path $\texttt{D} \cdot \texttt{X}$, which would lead to state 2 in the first automaton, and state 2 in the second automaton, and so on. The "–" state indicates the corresponding automaton cannot accept the current path or any extension of it. Since node (W,5,-) is in an accepting state for the first automaton, it indicates that this path has rank 1.

5.5.4 Product Graph Minimization

After building the PGIR, we subsequently minimize it. Minimizing the PGIR has several advantages including (1) the generated configurations will often be smaller, and (2) the smaller graph often improves the precision of our analysis that will infer local BGP preferences and whether a policy can be implemented safely under failures. The minimization is based on the observation that, although every path in the PGIR is a valid path in the topology, we do not want to consider paths that form loops. In particular, BGP's loop prevention mechanism forces an AS to reject any

route that already contains the AS in its AS path. For example, in Figure 5.9, the path `W·A·C·B·W` is a valid topological path, leading to a path that satisfies the preference 2 policy, but which contains a loop.

We use graph dominators [74] as a relatively cheap approximation for removing many nodes and edges in the PGIR that are never on any *simple* (loop free) path between the start and end nodes. In the PGIR, a node m dominates a node n if m appears on every path leading from the start node to n. Similarly, a node m post-dominates a node n in the PGIR if m appears on every path from n to the end node. We can safely remove nodes and edges in the PGIR when any of the following conditions hold, where we have m, m' and n, n' such that $m \approx m'$ and $n \approx n'$ (recall that $m \approx m'$ means that m and m' share the same topology location).

- Remove m if it is not reachable from the start node. In this case, BGP advertisements can not reach m so there is no need to consider its impact on the policy.
- Remove m if it can not reach the end node. In this case, m can never result in a path that leads to a policy-compliant path, so removing it will not impact the policy.
- Remove m if it is (post-)dominated by some m'. In this case, any advertisement leaving m will go through some m' where $m \approx m'$. This means that any path from m from the destination will form a loop so m can be thrown away.
- Remove edge (m, n) if some m' post-dominates n. If every path from n must go through m' after n receives an advertisement from m, then there is no loop-free path that is policy compliant after m, so this edge can be safely removed.
- Remove edge (m, n) if some n' dominates m. If any path to m must have gone through n', then we can delete the edge from m to n since this will always be on a path with a loop.

For example, node `(W,1,1)` in Figure 5.9 is removed because every path to the end node must always go through node `(W,-,4)`. That is, node `(W,1,1)` is post-dominated by node `(W,-,4)`

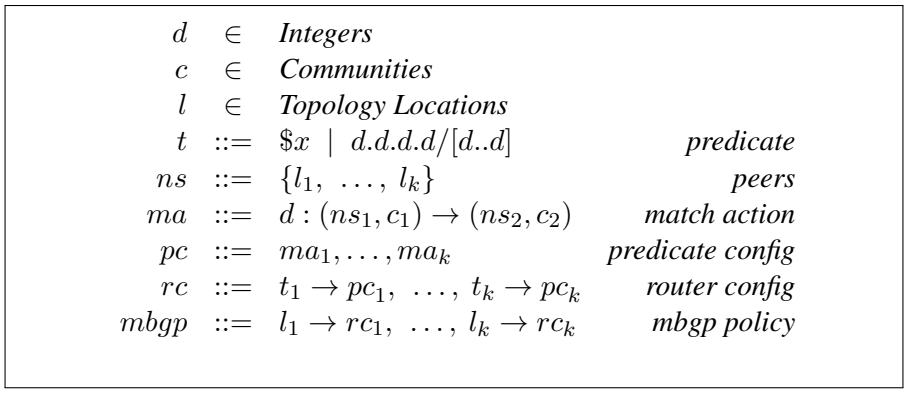


Figure 5.10: mBGP intermediate language syntax.

and both are shadows of topology location `W`. Similarly, the edge from `(C,3,2)` to `(D,-,2)` is removed since node `(C,-,2)` post-dominates `(D,-,2)`.

We repeatedly apply the minimizations above until no further minimization is possible. In the example from Figure 5.9, colored nodes and dashed edges show edges and nodes removed after minimization.

5.5.5 An intermediate BGP language

To define compilation we first introduce a simple, vendor independent configuration language for the BGP protocol called mBGP. The syntax for mBGP is shown in Figure 5.10 (left). An mBGP policy consists of a sequence of router configurations (one for each internal topology location l). A router configuration is an ordered sequence of pairs, where each pair contains a predicate describing the traffic, and a predicate configuration. A predicate is either a template variable `$x` (we will see this in Chapter 6) or a prefix. A predicate configuration is a collection of match action statements, where each match action indicates that the router will match advertisements from any of a set of peers ns_1 with a particular community tag c_1 with preference d, before exporting the route to another set of peers ns_2 with a new community tag c_2. The match action statements are applied in order from first to last, with later match actions only applying if a previous one has not already. The preference d indicates the priority of a message compared to other messages the

$$
\begin{array}{l}
\textbf{compile}_{\textbf{mBGP}}([(t_1, PG_1, pref_1), \ldots, (t_k, PG_k, pref_k)], G) = \\
\quad [\, l \rightarrow rc \mid l \in \text{internal}(G.V),\ rc = \text{append} \\
\qquad [\, t_i \rightarrow [\, ma \mid \\
\qquad\quad m \leftarrow (l, q_m) \in PG_i, \\
\qquad\quad (b, q_n) \leftarrow \text{adjIn}(PG_i, m), \\
\qquad\quad out \leftarrow \{c \mid (c, \_) \in \text{adjOut}(PG_i, m)\}, \\
\qquad\quad ma = \text{pref}_i(m) : (\{b\}, q_n) \rightarrow (out, q_m)\,]\,]\,] \\
\\
\textbf{compile}(p_1, \ldots, p_k, G) = \\
\quad \text{compile}_{\text{mBGP}}([\text{compile}_{\text{PG}}(p_1, G), \ldots, \text{compile}_{\text{PG}}(p_k, G)], G)
\end{array}
$$

Figure 5.11: Compilation from product graphs to mBGP.

router might received. The router will choose the route learned with the highest preference. The preference is left abstract, but would typically be implemented using the BGP local preference field. However, as we will see shortly, it can also be implemented using other BGP fields like the MED field.

Example: Consider the following mBGP policy:

```
A → (1.2.3.4/32 → 110:({B},1) → ({C,D},2))
```

This policy states that A will match any route advertisement with destination prefix `1.2.3.4/32`. If the message comes from B with tag 1, then A will assign it a preference of 110 and then export the route to neighbors `C` and `D` after updating the tag to be 2.

5.5.6 Compilation to mBGP

Figure 5.11 defines compilation from Propane to mBGP. It proceeds by compiling constraints p_i in the original Propane policy to a tuple of: the predicate t_i, the product graph PG_i, and a local preference function pref_i. For now we assume pref_i is given to us by an oracle. Section 5.6 will describe how we can compute this function. These tuples are passed to the $\text{compile}_{\text{mBGP}}$ function, along with the network topology G. For each internal router in the topology l, and each predicate t_i

in the Propane policy, compilation goes through each node m for l in PG_i, and groups the inbound neighbors of m by tag (q_n) into sets (in). It allows imports from these neighbors before exporting to the outbound neighbors of m. The local preference for these imports is given by $\mathrm{pref}_i(m)$. We build configurations using list-comprehension notation. For instance, $[\ l \rightarrow rc \mid l \in V, p(l, rc)\ ]$ denotes the mBGP policy $l_1 \rightarrow rc_1, ..., l_k \rightarrow rc_k$ where each l_i is drawn from V, and each rc_i satisfies $p(l_i, rc_i)$. We use the function append to denote sequence concatenation.

The idea behind the translation to mBGP is straightforward. Namely, we encode the state of the automata using BGP community values. Each router will match based on its peer and a community value corresponding to the state of the PGIR, and then update the state before exporting to the neighbors permitted by the PGIR. The function $\mathrm{compile}$ takes as input a collection of propane policies of the form $p_i = t_i$ => $r_{i1} \ldots r_{in}$ mapping predicates to regular paths and preferences. It then creates product graphs for each such p_i and calls $\mathrm{compile}_{\mathrm{mBGP}}$ on the collection of product graphs along with their original policies. $\mathrm{compile}_{\mathrm{mBGP}}$ goes through each product graph and policy (in order of priority), and then walks through each internal router in the network. For each router and product graph pair, it picks out all the PG nodes corresponding to that router ($m \leftarrow (l, q_n)$. It then looks at all the inbound PG neighbors ((b, q_n) with edges going to m) as well as all the outbound PG neighbors ($(c, \_) \in \mathrm{adjOut}(PG, m)$). For each such combination, it creates a new rule that matches from the inbound neighbor ((b, q_n)) and then exports to all outbound neighbors. The preference set for such a match is given by the inferred preference function pref_i for each product graph PG_i. All the results are then concatenated into a single, final list of rules corresponding to the policy for that predicate (t_i).

Example: Figure 5.12 shows the generated configurations for the running compilation example. In the generated configurations, `A` will allow an announcement from `C` with a community value for state `(3,2)` (and deny anything else). If it sees such an announcement, it will remove the old community value and would a new one for state `(4,2)` before exporting it to `W`. However, because `W` is an external node, a final post-processing pass would recognize that the noexport community

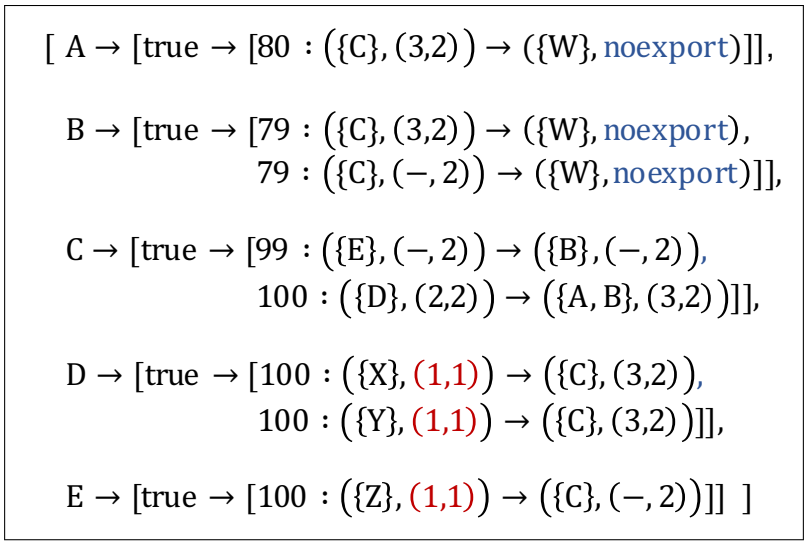


Figure 5.12: Generated mBGP router configurations.

needs to be attached to `W` to ensure the correct behavior, so noexport replaces `(4,2)`. Similarly, `B` will accept routes from `C` in either state before sending them to `W`. Router `C` will accept routes from either `E` or `D` and then export to either `B` or both `A` and `B` accordingly. It will also update the community tag depending on which match was triggered. For each router r, the compiler sets a local preference according to the pref_i function. For `A` and `B`, this preference corresponds to the MED value that will influence `W`. For `C`, this will be the local preference. Specifically, `C` will prefer an advertisement from `D` in state `(2,2)` over an advertisement from `E` in state `(-,2)` because it uses a lower (worse) local preference for routes from `E`.

Since the compiler can control community tagging only for routers under the control of the AS being programmed, it cannot match on communities for external ASes. Such communities are shown in red. Instead, a final post-processing phase translates matches from external AS communities into a BGP regular expression filter. For example, node `D` in Figure 5.9 would match the single hop external paths `X` or `Y`. In general, if routes are allowed from beyond `X` or `Y`, these will also be captured in the BGP regular expression filters. The unknown AS topology is modeled as a special node in the PGIR that generates a filter to match any sequence of ASes.

Finally, the external AS W should prefer our internal router A over B. In general, it is not possible to reliably control traffic entering the network beyond certain special cases. In this example, however, assuming our network and W have an agreement to honor MEDs, the MED attribute can influence W to prefer A over B. Additionally, the compiler can use the BGP no-export community to ensure that no other AS beyond W can send us traffic. The compiler performs a simple analysis to determine when it can utilize BGP special attributes to ensure traffic enters the network in a particular way by looking at links in the product graph that cross from the internal topology to the external topology.

5.5.7 Configuration Minimization

In addition to minimizing the PGIR, after configuration generation, to shrink configurations and make them easier to understand for a human, the compiler further processes the mBGP policy in a number of ways. First, it translates global community values into local ones to reduce the number of community tags needed to implement the policy. The combination of the neighbor through which a router receives an advertisement and the local tag from the neighbor is enough to unambiguously recover the global tag. This allows reuse of the same community tags across all routers. Next, it removes any unused community tags when they are not needed. For example, if the policy has only a single unique community tag, then it need not be added at all. Similarly, if the compiler can unambiguously recover the automata state just from knowing the neighbor through which the advertisement is received, then it does not need to tag that route at that node.

Yet another minimization optimization is to combine filters when possible. If a router has the same filters to multiple neighbors, then they can be grouped into a reusable policy route-map that will reduce code duplication. Similarly, if a filter is equivalent to the empty filter, then the compiler can remove a filter entirely from an interface.

In the mBGP policy shown in Figure 5.12, all community tags can be removed, since there is never any ambiguity as to the current state of the PGIR based only on the current router importing

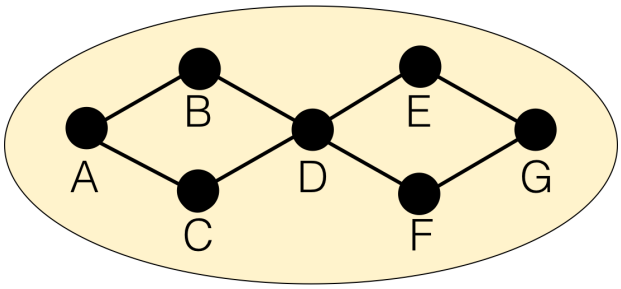


Figure 5.13: A network where the policy $(\texttt{A} \cdot \texttt{B} \cdot \texttt{D} \cdot \texttt{E} \cdot \texttt{G}) >> (\texttt{A} \cdot \texttt{C} \cdot \texttt{D} \cdot \texttt{F} \cdot \texttt{G})$ is unimplementable in BGP under arbitrary failures.

the route and the neighbor from which the route is being imported. As with PGIR minimization, we repeatedly apply the different minimization passes until no further minimization is possible.

5.6 Preference Inference

The compilation scheme previously described will ensure that BGP only ever searches through paths that are allowed by the policy (*i.e.*, by tagging routes with community values corresponding to automata states and dropping routes corresponding to disallowed paths). However, how can the compiler determine what local preferences each router should assign to different routes to ensure that they coordinate enough to always find the *best* allowed path rather than just *some* allowed path? This task becomes even more challenging in the presence of failures since routers running BGP lack a global view of the network. To illustrate why this is challenging we consider what can happen when different failures might occur in a network.

The problem with failures: Consider the simple policy for the topology in Figure 5.13, which says to prefer the top path over the bottom path: $(\texttt{A} \cdot \texttt{B} \cdot \texttt{D} \cdot \texttt{E} \cdot \texttt{G}) >> (\texttt{A} \cdot \texttt{C} \cdot \texttt{D} \cdot \texttt{F} \cdot \texttt{G})$. Recall that routing advertisements flow in the opposite direction of traffic (starting from G). How could such a policy be implemented in BGP? Suppose we set the local preferences to have D prefer E over F, and have A prefer B over C. This works as expected under normal conditions, however, if the A–B link fails, then suddenly D has made the wrong decision by preferring E. Traffic will now follow the $\texttt{A} \cdot \texttt{C} \cdot \texttt{D} \cdot \texttt{E} \cdot \texttt{G}$ path, even though this path was not allowed by the policy. Thus,

the distributed implementation has used a route that is not allowed by the policy. To make matters worse, the second preference for the path A · C · D · F · G is available in the network but not being used. Thus, a path for the best possible route available after the A–B failure exists in the network, but the distributed implementation will not find it. The first problem could be fixed by tagging and filtering route advertisements appropriately so that C rejects routes that go through E, however the second problem cannot be fixed. In fact, this policy cannot be implemented in BGP in a way that is policy compliant under all failures since D cannot safely choose between E and F without knowing whether the A–B link is available.

Preference Search: To enforce correct path preferences, we use the BGP local-preference and MED attributes, which allows routers to (locally) prefer certain routes (e.g., those from a particular neighbor or with a tag) over other routes. The challenge is to find a collection of device-local preferences that correctly enforce the policy's network-wide preferences in the face of any set of failures, or deduce that no such collection exists. The idea is to search for such a device-local preference function for each router that totally orders advertisements from different neighbors (possibly with different tags). For example, in Figure 5.9, to satisfy the policy that we prefer going through a path with the link C–D, router C should prefer advertisements tagged with (2,2) from D over those tagged with (-,2) from E.

In general, because BGP is distributed, each router does not have a view of the entire network when choosing which path to use, and finding a collection of preferences to ensure correct end-to-end behavior for all failures is a hard problem. Instead, we introduce a conservative search strategy for determining route preferences that we have found works well in practice. In particular, the search strategy identifies a particular type of PGIR graph structure that is present for most well-formed policies, and the search strategy is based on finding a ranking of PG nodes for each concrete node that indicates the relative preference that should be used when importing from neighbors of that PG node. To accomplish this, we define the following (recall that $\mathrm{rank}(n) = \{i \mid i \in F_i\}$ picks out the regular preferences that are in an accepting state at node n):

Definition 5.6.1. *Let $m \geq_{rank} n$ be a relation over PG vertices that holds iff $m \approx n$ and either* $\min(\text{rank}(m)) \geq \min(\text{rank}(n))$ *or* $\text{rank}(n) = \emptyset$.

Intuitively $m \geq_{rank} n$ means that paths ending at node n have lower automata rank and are thus better than paths ending at m.

Labelled transition system: The PG can be viewed as a labeled transition system by pushing the location from each directed edge's target node onto the edge. That is, $m \xrightarrow{l} n$ if there is an edge (m, n) in the PG and $\tilde{n} = l$. For example, Figure 5.9 we have the transition `(C,3,2)` $\xrightarrow{A}$ `(A,4,3)`.

Definition 5.6.2. *We write $m \leq n$ if the subgraph reachable from m and n respectively form a simulation relation with respect to $\geq_{rank}$. In other words, we say that $m \leq n$ if $n \geq_{rank} m$ and for every transition $n \xrightarrow{l} n'$ from PG node n there exists a transition $m \xrightarrow{l} m'$ from m and $m' \leq n'$.*

If $m \leq n$, then advertisements received for node m can safely be preferred over those received for node n after accounting for the network-wide impact of the choice. For example, in Figure 5.9, C can receive advertisements in two different contexts `(C,-,2)` and `(C,3,2)` and must choose between messages received in these different contexts. Advertisements are preferred in state `(C,3,2)` because they result in a better path for C (`(C,3,2)` $\geq_{rank}$ `(C,-,2)`) and downstream routers such as `A` will also obtain paths no worse than if `C` had chosen `(C,-,2)`.

Failure safety: This simulation relation also gives us a strong guarantee about safety. Namely, a policy that sets preferences according to this simulation relation ($\leq$) is guaranteed to be safe from failures because, if a link fails in the topology, then $m \leq n$ will still hold since any transition that becomes unusable for m also becomes unusable for n. That is $m \leq n$ before the failure implies $m \leq n$ after the failure. For each router, its corresponding PG nodes are sorted according to $\leq$. If the operator defines a total order on PG nodes, then the compiler can simply prefer advertisements from peers of node m over those of n whenever $m \leq n$. However, if $\leq$ does not form a total order, then the policy is rejected as being possibly unsafe under some failure conditions (since inference is conservative).

Algorithm 2 Inferring preferences

```
1:  procedure IsPreferred(G, N_1, N_2)
2:      if N_1 ≉ N_2 then return false
3:      q ← Queue()
4:      q.Enqueue(N_1, N_2)
5:      while !q.Empty() do
6:          (m, n) ← q.Dequeue()
7:          if m ⋦_rank n then return false
8:          for n' in adj(G, n) do
9:              if (∃m' ∈ adj(G, m), m' ≈ n') or
10:                 (∃m' ∈ G, dominates m, m' ≈ n') then
11:                 if (m', n') not marked then
12:                     mark (m', n') as seen
13:                     q.Enqueue(m', n')
14:             else return false
15:     return true
```

In Figure 5.9 the inequality `(C,3,2)` $\leq$ `(C,-,2)` holds since nodes on the left side of the PG can always match transitions made on the right hand side with respect to the $\geq_{rank}$ relation. This relation does not hold the other way around since `(C,-,2)` $\not\geq_{rank}$ `(C,3,2)`. Therefore, advertisements received at `C` with tag `(3,2)` must be preferred to those received at `C` with tag `(-,2)`.

A preference inference algorithm: Algorithm 2 checks whether one PG node can be preferred to another (with the same topology location). That is it determines if $N_1 \leq_{lp} N_2$. It walks from nodes N_1 and N_2 and ensures that for every *step* N_2 can take to some new topology location, N_1 can, at the very least, also take a step to an equivalent topology location ($\approx$). As an optimization, when there is no such equivalent step, the algorithm attempts to take into account where the advertisement must have already traversed. In particular, it checks if there is an equivalent dominator node and, if so, walks from this new node instead. The idea is that, since the advertisement must have already passed through the dominator, we can check to see if we are guaranteed to find paths that are at least as good from this new node instead. At each step, it requires that the current node reachable from N_1 has a path rank that is at least as good as that of the current node reachable from

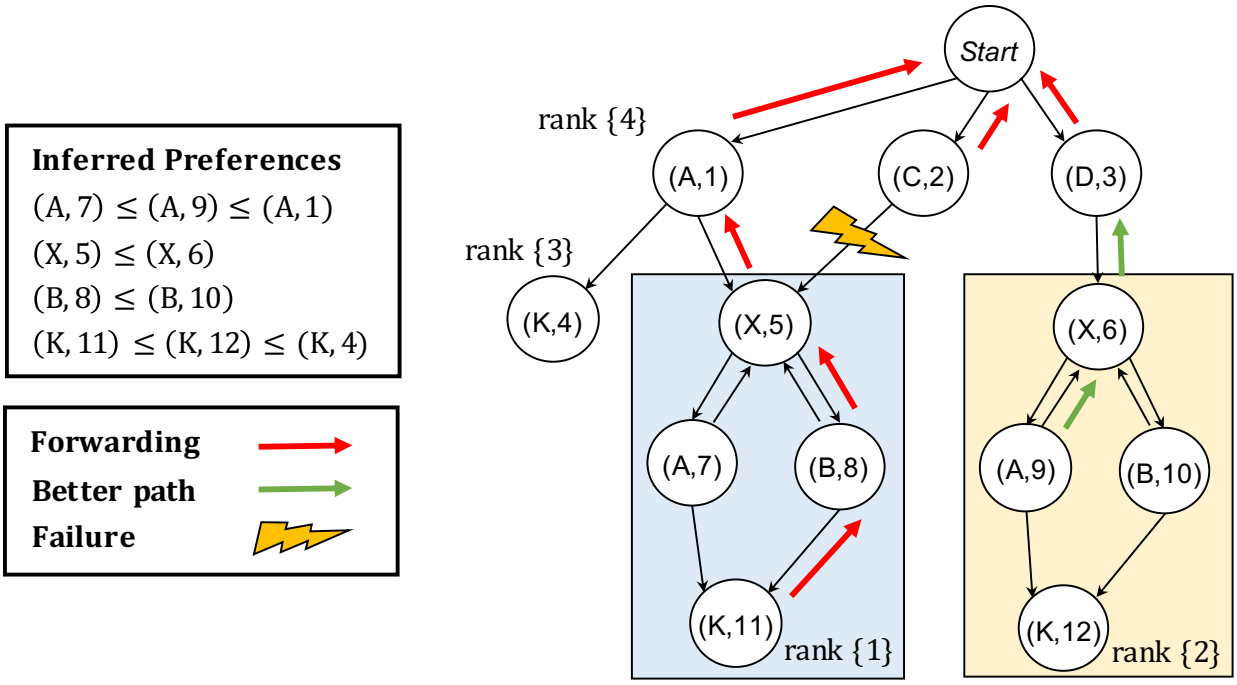


Figure 5.14: Product graph where preference inference is unsound due to loops.

N_2 ($m \leq_{rank} n$). The intuition here is that if $m \not\leq_{rank} n$, then we can very likely fail every edge in the topology except for the path that leads to the current m and n, thereby generating a counterexample. Algorithm 2 terminates since the number of related states (m, n) that can be explored is finite.

For each router in the topology, local preferences are now obtained by sorting the corresponding PGIR topology locations according to the $(\leq)$ relation determined by Algorithm 2. If two nodes are incomparable, then the compiler rejects the policy as unimplementable. In the example, because we compute `(C,3,2)` $\leq$ `(C,-,2)`, we must assign `(C,3,2)` a better (higher) local preference than `(C,-,2)`.

5.6.1 Avoiding loops

The checks for failure safety described above overlook one critical point: A better (lower rank) path might not be available due to loops rejected by BGP. Consider the product graph shown in Figure 5.14. Nodes in the blue square (*e.g.*, `(X,5)`) have rank 1, while similar nodes on the right

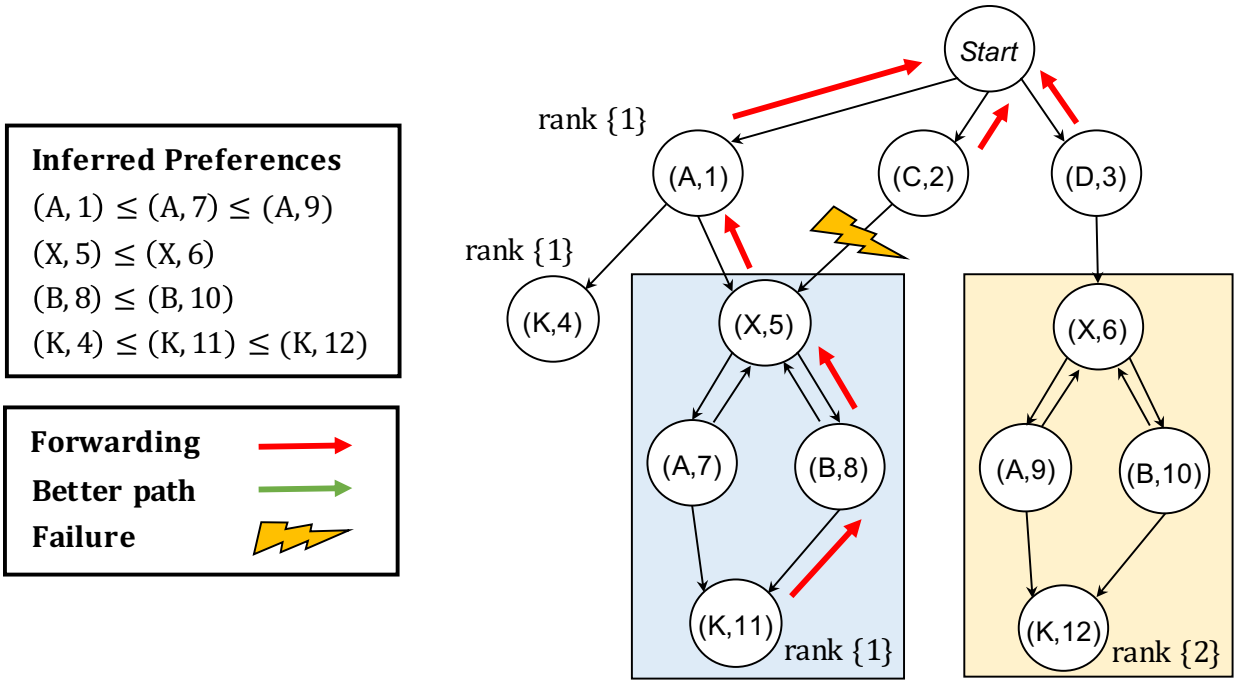


Figure 5.15: A similar product graph where preference inference is sound.

(*e.g.*, `(X,6)`) have rank 2. Node `(A,1)` has a higher (worse) rank of 4 and node `(K,4)` has a rank of 3. The inferred local preferences for the nodes are shown on the left. For instance, the preference inference algorithm will find that `(X,5)` is preferred to `(X,6)` because it satisfies the simulation relation. Similarly, `(A,7)` can be preferred over `(A,9)` and `(A,1)`.

However, when applying Algorithm 2, we failed to take into account the possibility of loops. Suppose that the topology link between `X` and `C` fails. The resulting forwarding behavior is shown going the opposite direction (in red) superimposed on top of the product graph. `X` will select a route through `A` (via `(A,1)`) before propagating this message to its neighbors. However, when `A` receives a message in state `(A,7)` from `X`, this route will be discarded due to BGP loop prevention since the path already contains `A`. Thus, `A` will end up with a path from `(A,1)`. Unfortunately though, there is a better path available for `A` via `(A,9)`. However the network will never find this path since `X` will not select a route in state `(X,6)`. Therefore, this BGP implementation is *not* policy compliant for all failures.

The problem was that, when performing the preference inference, we ignored the possibility that a better path (*e.g.*, through `A` in the blue) might not be possible due to loops (*e.g.*from `(A,1)`).

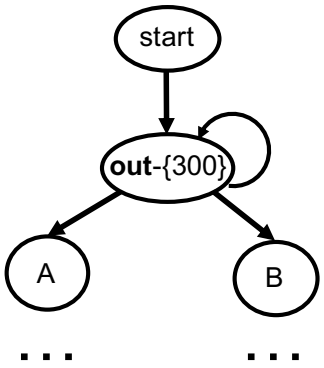


Figure 5.16: Representing external nodes in the product graph

On the other hand, this would not have been a problem if paths ending at `(A,1)` had a lower rank than those ending at `(A,7)` or `(A,9)`.

For example, consider a slightly different product graph in Figure 5.15. Now paths ending at `(A,1)` and `(K,4)` now have a better rank of 1 and the inferred preferences are changed accordingly. The difference is that now any time `(A,7)` is unusable due to a loop with `(A,1)`, it ultimately does not matter since `(A,1)` is preferred anyway. In fact, checking if we are never worse off using `(A,1)` instead of `(A,7)` corresponds exactly with determining if `A` has can prefer `(A,1)` over `(A,7)`. More specifically, the compiler checks that, any time there are two nodes N_1 and N_2 for the same topology location, where N_1 appears "above" (*i.e.*, can reach) N_2 in the PGIR, then N_1 must be strictly preferred to N_2 (*i.e.*, $N_1 <_{\text{rank}} N_2$).

5.6.2 Modeling the rest of the Internet

One important point glossed over in the example, is that policies can refer to external ASes that may not be directly connected to the network being configured. As an example, consider the following policy, which rejects paths going through an AS in China:

```
define China = as300
define Reject = !through(China)
...
```

The difficulty with this policy is that matching AS paths that have gone through this AS can not be done simply by looking at a neighbor. In particular, it requires using a BGP regular expres-

sion match expression. To be able to compile such expressions, we include a special "outside" node in the PGIR that can represent any possible external AS minus some set of known locations. Figure 5.16 shows an example of a product graph for this policy. The special node **out** $-\{300\}$ represents any external node except for AS 300. There is a transition for this node to itself to represent the fact that it can represent any number of such nodes before an advertisement enters the network. When compiling a RIR policy then, we translate the negation of a set of locations `ls` into **out**$-$ `ls`.

Finally, to compile the policy to mBGP for router `A` then, instead of matching on the exact neighbor, we instead generate a BGP regular expression based on the product graph structure by using a standard automata-to-regex construction algorithm for the reverse graph starting from `A`. In this case, `A` would generate a regular match of the form: `(.*)` $\cdot$ {**out**-{300}} $\cdot$ `(.*)`, which then would be translated into a vendor-specific regex format.

5.7 Safety Analysis

5.7.1 Aggregation-safety Analysis

Aggregation can lead to subtle black-holing of traffic when failures occur. Determining when this can happen requires knowledge, not only of the topology, but also of the policy. For instance, a policy might require that all traffic for a particular prefix go over a single link before being aggregated. If that one link fails, a black hole might be introduced. Because the PGIR encodes the complete user policy and topology, Propane can efficiently check that aggregates do not black hole traffic for up to k failures.

We view the aggregation problem as a variant of the min-cut problem in the PGIR. Specifically, for each prefix that falls under an aggregate, we are interested in finding a lower bound on the number of failures required to disconnect the prefix's origin from its aggregation point. The difficulty, however, is that each link in the topology might appear as multiple links in the PGIR, thus preventing the direct application of standard min-cut algorithms.

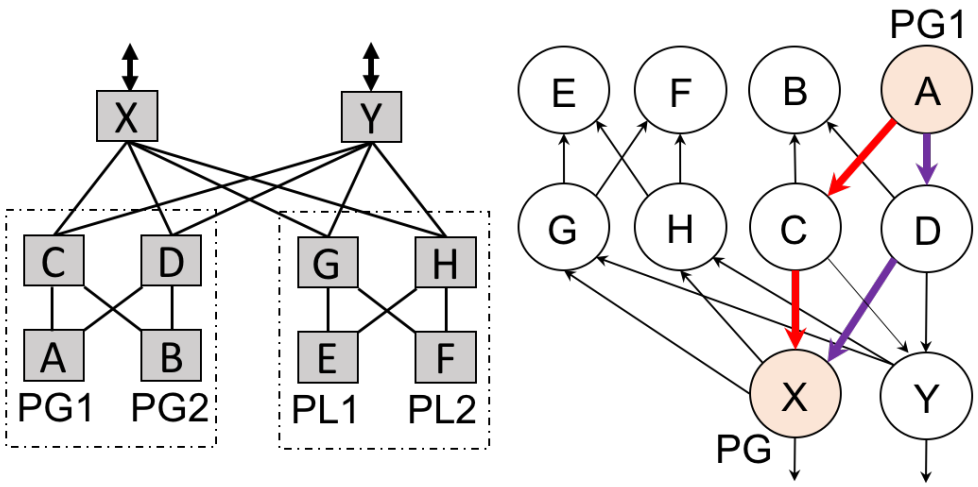


Figure 5.17: Aggregation safety for a datacenter.

Instead, we adopt the following simple strategy: (1) pick a random path in the PGIR between the prefix's origin and aggregation point, (2) remove all edges between the same topology locations (edge source and destination) in the PGIR for each edge along the chosen path, and (3) repeat until no such path exists. Because each path chosen is both policy compliant and edge disjoint (due to 2), the number of paths that we are able to remove lower bounds the number of failures required to disconnect the prefix from its aggregate, subject to the policy constraints.

Recall the datacenter example from Section 5.3, with the policy `PG1 => end(A)`, where `PG1` falls under the `PG` aggregate. Figure 5.17 shows the PGIR for `PG1`. Since we know aggregation will occur at `X`, and that the `PG1` prefix will originate at `A`, we can compute the number of failures it would take to disconnect `A` from `X`. We could remove the `A–D–X` path first and would then need to remove any other `A–D` or `D–X` links from the PGIR (in this case none). Next, we could remove the links along the `A–C–X` path, repeating the process. Because `A` is now disconnected from `X`, 2 is a lower bound on the number of failures required to introduce an aggregation black hole for prefix `PG1`. This process is repeated for other aggregation locations (*e.g.*, `Y`).

5.7.2 Other Analyses

Checking policy correctness: Even when programming the network centrally, it is possible for operators to make mistakes. Propane includes many analyses to identify common mistakes at

compile time. A subset includes: (1) a preference analysis to determine when backup paths will never be used, (2) a reachability analysis to check if locations that should be reachable according to the policy are not reachable after combining the topology and policy, (3) an anycast analysis to find instances where the operator might accidentally anycast a prefix (*i.e.*, originates the prefix from multiple locations), (4) an aggregate analysis to find unused aggregates that do not summarize any specific prefix.

5.8 Implementation

Our Propane compiler is implemented in roughly 9000 lines of F# code [10]. It includes command-line flags for enabling or disabling the use of the BGP MED attribute, AS path prepending, the no-export community, and for ensuring at least k-failure safety for aggregate prefixes. Since each prefix predicate has a separate routing policy, we compile each routing policy in parallel. Currently, Propane supports generating Quagga router configurations out of the box. Users can add new vendor-specific adapters to translate from mBGP to other router configuration languages, or incorporate the compiler into an existing template-based system, *e.g.*, by mixing the Propane-generated BGP configuration with other, non-BGP configuration elements.

Our compiler has the following features that improve its performance and usability.

Efficient PGIR construction: Constructing automata for extended regular expressions (*i.e.*, regular expressions with negation and intersection operations) is known to have high complexity [48]. The Propane compiler uses regular expression derivatives [83] with character classes to construct deterministic automata for extended regular expressions efficiently. Since regular expressions are defined over a finite alphabet, and since much of the AS topology is unknown, we set the alphabet to include all uniquely referenced external ASes in the policy. Rather than construct the product graph in full, our implementation prevents exploring parts of the graph during construction when no automata has a reachable accepting state.

PG Minimization: The Propane compiler uses a fast algorithm for computing graph dominators by storing sets of dominators in a compact representation known as a dominator tree [74]. All minimization steps are repeated until no more progress can be made.

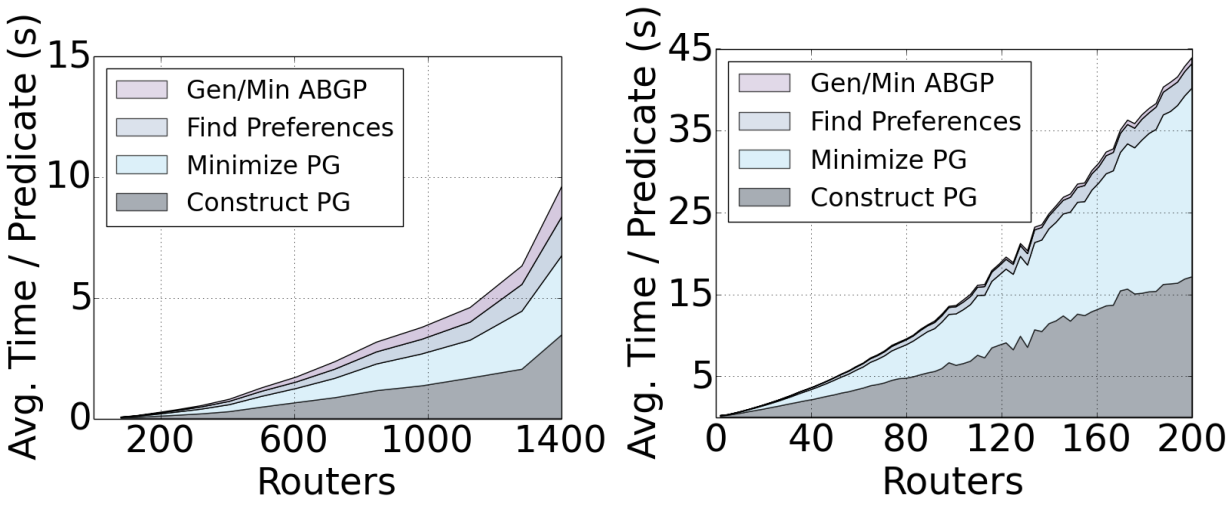
Fast failure-safety analysis: When computing local preferences and ensuring failure safety, as described in Section 5.5, the compiler performs memoization of the IsPreferred function. That is, whenever for two states N_1 and N_2 we compute $\mathrm{IsPreferred}(G, N_1, N_2)$ and the function evaluates to $true$, then each of the intermediate related states m and n must also satisfy $\mathrm{IsPreferred}(G, m, n)$. Memoizing these states dramatically reduces the amount of work performed to find preferences in the common case.

Efficient configuration generation: The naive code generation algorithm described in Section5.5 is extremely memory inefficient since it generates a separate match-export pair for every unique in-edge/out-edge pair for every node in the product graph before minimization. Our implementation performs partial minimization during generation by recognizing common cases such as when there is no restriction on exporting to or importing from neighbors.

5.9 Evaluation

We apply Propane on real policies for backbone and datacenter networks. Our main goals are to evaluate if its language is expressive enough for real-world policies, the time the compiler takes to generate router configurations, and the size of the resulting configurations.

Networks studied: We obtained routing policy for the backbone network and datacenters of a large cloud provider. Multiple datacenters share this policy. The backbone network connects to the datacenters and also has many external BGP neighbors. The high-level policies of these networks are captured in an English document which guides operators when writing configuration templates



(a) Datacenter

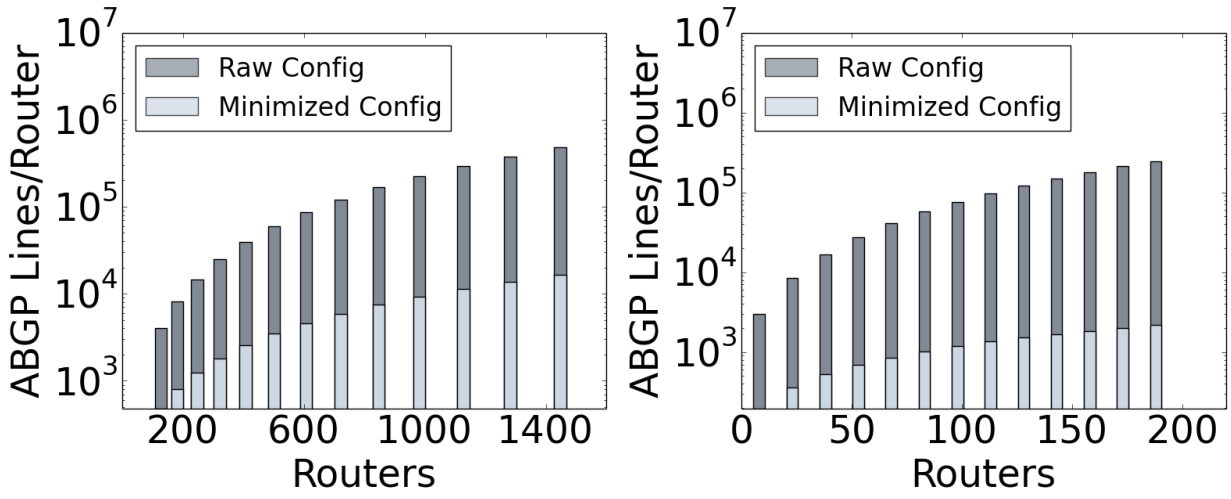
(b) Backbone

Figure 5.18: Compilation time.

for datacenter routers or actual configurations for the backbone network (where templates are not used because the network has a less regular structure).

The networks have the types of policies that we outlined earlier (Section 5.3). The backbone network classifies external neighbors into several different categories and prefers paths through them in order. It does not want to provide transit among certain types of neighbors. For some neighbors, it prefers some links over the others. It supports communities based on which it will not announce certain routes externally or announce them only within a geographic region (*e.g.*, West Coast of the USA). Finally, it has many filters, *e.g.*, to prevent bogons (private address space) from external neighbors, prevent customers from providing transit to other large networks, prevent traversing providers through peers, *etc.*

Routers in the datacenter network run BGP using private AS numbers and peer with each other and with the backbone network over eBGP. The routers aggregate some prefix blocks when announcing them to the backbone network, they keep some prefixes internal, and attach communities for some other prefixes that should not traverse beyond the geographic region. The datacenter networks also have policies by which some prefixes should not be announced beyond a certain tier in the datacenter hierarchy.



(a) Datacenter **(b)** Backbone

Figure 5.19: Configuration minimization.

Expressiveness: We found that we could translate all network policies to Propane. We verified with the operators that our translation preserved intended semantics. We also asked the two operators if they would find it easy to express their policies in Propane. The datacenter operator said that he found the language intuitive. The backbone operator said that formalizing the policy in Propane seemed equally easy or difficult as formalizing in RPSL [4], but he appreciated that he would have to do it only once for the whole network (not per-router) and did not have to manually compute various local preferences, import-export filters, and MEDs. For the backbone network, the operator mentioned an additional policy not present in the English document, which we added later. For both the datacenter and backbone networks, Propane was able to guarantee policy-compliance under all possible failure scenarios.

Not counting the lines for various definitions like prefix and customer groups or for prefix ownership constraints, which we cannot reveal because of confidentiality concerns, the routing policies for Propane were 43 lines for the backbone network and 31 lines for the datacenter networks.

Compilation time: We study the compilation of time for both policies as a function of network size. Even though the networks we study have a fixed topology and size, we can explore the impact of size because the policies are network-wide and the compiler takes the topology itself as an input.

For the datacenter network, we build and provide as input fat tree [3] topologies of different sizes, assign a /24 prefix to each ToR switch, and randomly map prefixes to each type of prefix group with a distinct routing policy. For the backbone network, the internal topology does not matter since all routers connect to each other through iBGP. We explore different (full iBGP) mesh sizes and randomly map neighboring networks to routers. Even though each border router connects to many external peers, we count only the mesh size.

All experiments are run on an 8 core, 3.6 GHz Intel Xeon processor running Windows 7. Figure 6.14 shows the compilation times for datacenter and backbone networks of different sizes. For both policies, we measure the mean compilation time per prefix predicate since the compiler operates on each predicate in parallel. A single predicate can describe many prefixes, for example by matching on a disjunction of prefixes. At their largest sizes, the per-predicate compilation time is roughly 10 seconds for the datacenter network and 45 seconds for the backbone network.

Compilation for the largest datacenter takes less than 9 minutes total. Unlike the datacenter policy, the number of predicates for the backbone policy remains relatively fixed as the topology size increases. Compilation for the largest backbone network takes less than 3 minutes total. The inclusion of both more preferences and more neighboring ASes in the backbone policy increases the size of the resulting PGIR, which in turn leads to PGIR construction and minimization taking proportionally more time.

In both examples, we observe that Algorithm 2 for inferring preferences is efficient, taking only a small fraction of the total running time. PGIR minimization is the most expensive compilation phase. If needed, minimization can be limited to a fixed number of iterations for large networks. Both the backbone and datacenter policies could be successfully compiled without performing minimization.

Configuration size: Figure 5.19 shows the size of the compiled mBGP policies as a function of the topology size. The naive translation of PGIR to mBGP outlined in Section 5.5 generates extremely large mBGP policies by default. To offset this, the compiler performs mBGP configura-

tion minimization both during and after the PGIR to mBGP translation phase. Such minimization is useful for limiting the computational expense of matching routes on BGP routers, reducing the number of forwarding entries in routers in certain cases, and making configurations more readable for humans. Minimization is highly effective for both the datacenter and backbone policies. In all cases, minimized policies are a small fraction of the size of their non-minimized counterparts.

However, even minimized configurations are hundreds or thousands of lines per router. For the backbone network, the size of Propane configurations is roughly similar to the BGP components of actual router configurations, though qualitative differences exist (see below). We did not have actual configurations for the datacenter network; they are dynamically generated from templates.

Propane vs. operator configurations: We comment briefly on how Propane-generated configurations differ from configurations written by operators. In some ways they are similar. For example, preferences among neighboring ASes are implemented with a community value to tag incoming routes according to preference, which is then used at other border routers to influence decisions.

In other ways, the Propane configurations are different, relying on a different BGP mechanism to achieve the same result. Some key differences that we observed were:

1. operators used the no-export community to prevent routes from leaking beyond a certain tier of the datacenter, while Propane selectively imported the route only below the tier; we believe Propane could use a similar implementation mechanism in the future as an optimization.

2. operators prevented unneeded propagation of more-specific route announcements from a less-preferred neighboring AS based on their out-of-band knowledge about the topology, whereas Propane propagated these advertisements;

3. operators used a layer of indirection for community values, using community groups and rewriting values, to implement certain policies in a more maintainable manner, where Propane uses flat communities; and

4. operators used BGP regular expression filters to enforce invariants that are independent of any particular prefix, whereas Propane enforced these invariants per prefix.

5.10 Summary

In this chapter, we introduced the idea of network control plane synthesis. Like verification, synthesis aims to improve network reliability by reducing the likelihood for bugs. However, while verification addresses the problem of checking existing configurations for correctness, synthesis is responsible for generating correct-by-construction configurations directly from a high-level specification. To make synthesis possible, we introduced two new ideas: (1) a new high-level language called Propane for writing down network policies, and (2) a way to compile Propane policies down to the BGP routing protocol. The key abstraction in Propane is the ability to write end-to-end network policies using regular expressions and path preferences rather than writing individual device-local policies. The compiler is then responsible for turning this end-to-end policy into something that can be implemented in a distributed way. Our empirical evaluation suggests that Propane can capture many real policies and can be compiled relatively quickly for large networks.

Chapter 6

Control Plane Synthesis with Abstraction

As with verification, a natural question is whether or not we can apply the idea of network abstraction to the problem of synthesis. That question is the subject of this chapter. While not slow, synthesis of BGP configurations for larger networks with 1000+ routers can still take Propane minutes, and as the topology size increases, the time it takes to perform synthesis grows super-linearly. For synthesis in particular, there is an additional benefit to reasoning over an abstract representation; A single abstract network can represent multiple concrete networks simultaneously. By reasoning directly over the abstract network we can reason about all possible concrete instantiations simultaneously. Such reasoning allows network operators, for example, to change and evolve their networks over time in certain prescribed ways and know that the old configurations will still work as expected.

6.1 Overview

To understand why abstraction is useful for synthesis, it is helpful to consider how operators manage complexity in configuration today. Specifically, to configure large networks, instead of considering individual devices, operators will often classify devices into *roles* where a role refers to specific functionality and is served by one or more devices. For instance, in a data center, roles may be "top-of-rack," "aggregation," and "spine" routers; and in a backbone network, they may

be "core" and "border" routers. While the network may have hundreds or thousands of devices—a scale that is impossible for humans to handle—there tend to be only a handful of roles. Operators often author a configuration *template* for each role. Templates are macros that, given a network topology, can be instantiated with different concrete values to generate device configurations.

Unfortunately, templates use the same low-level constructs as ordinary router configurations (*e.g.*, adding or removing tags from announcements). To validate their templates, operators will typically first instantiate a template with appropriate concrete parameters and then test it under various scenarios. Like normal configurations, such testing is inherently incomplete, but templates introduce additional complexity since different concrete parameters can lead to different behavior. Moreover, even if network operators were to instantiate their templates using the initial network topology and verify key properties using tools such as Minesweeper, the guarantees would not necessarily hold as the network topology evolves. Evolution of the topology is a frequent event for large networks, as devices and links are taken offline for maintenance and added to expand capacity. Templates that work for the current topology may or may not work for future topologies. Ensuing problems may cause operators to make non-uniform changes to routers' configurations, which defeats the purpose of a template system. An even worse situation is when operators must update many devices to evolve their network. Such network-wide configuration changes entail a great deal of risk and can be highly disruptive to live traffic.

While templates introduce many challenges and problems, they also very naturally capture the hierarchical structure of the network that operators find intuitive (*e.g.*, thinking about the "spine" role rather than individual routers). Our conversations with two major cloud providers reveal that operators of large networks are reluctant to use synthesis tools because, while they think of their network abstractly in terms of roles, synthesis tools like Propane operate over concrete topologies. Even if two devices play the same role, operators cannot specify policy in terms of this role; and even if specifications for the two devices are similar, there is no guarantee that the systems will generate (syntactically) similar configurations. Perhaps most importantly, if the operators want to debug or analyze system output, they will have to consider hundreds of device configurations

instead of just a handful of role configurations. Current synthesis systems are also brittle in the face of network evolution. Any change in network topology requires re-execution of the engine, from scratch, on the new topology, and the result may be a completely different set of configurations. No operator can shut down a large, production network, upgrade policy on all devices and then restart their network.

Our approach: To address the challenge of configuration synthesis in the presence of abstract roles, we develop Propane/AT. Propane/AT allows operators to input *abstract topologies* in terms of roles and their connectivity. For instance, they may specify roles for "top-of-rack" and "aggregation" routers and specify that every top-of-rack router connects to at least two aggregation routers (to tolerate the failure of a link to an aggregation router). Besides an abstract topology, Propane/AT takes two additional inputs. The first is a high-level specification of routing policy in the style of Propane. While Propane policies refer to concrete devices, Propane/AT policies refer to abstract roles. The final input to Propane/AT is the fault-tolerance requirements of the network, such as the number of simultaneous link failures it can tolerate without loss of connectivity for any traffic flow.

Based on these inputs, Propane/AT generates one template per role. These templates specify routing policy using BGP. The templates are correct for *any* concrete topology that complies with (*i.e.*, is a valid concrete instance of) the abstract topology. They are also evolution friendly. When the network evolves from one compliant concrete topology to another, only the configurations of devices that acquire or lose a neighbor need to change. This guarantee is the best that any system can give as neighbor relationships are explicitly configured in devices. We achieve it in part by using a form of source-routing with BGP – that is, expressing policy using such tags (instead of router- and prefix-based identifiers).

During synthesis, the Propane/AT compiler analyzes abstract topologies to determine the fault tolerance properties of the specified routing policy. This analysis yields a lower bound on the number of link failures required to disconnect one abstract location from another – hence any concrete

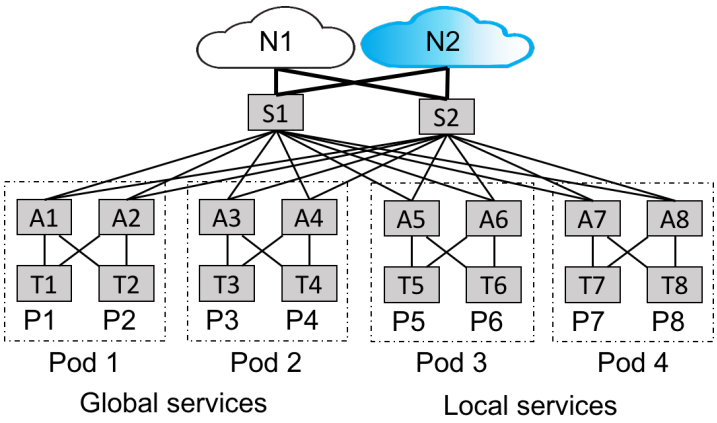


Figure 6.1: An example data center network.

instance of the abstract topology will adhere to the given fault tolerance property. We show how fault-tolerance analysis can be done in the abstract domain by using a set of sound inference rules to infer a lower bound on the number of edge-disjoint paths in any concrete topology. Our analysis uses a set of sound inference rules and an optimizing SMT solver.

6.2 Configuration Templates

Consider the data center example in Figure 6.1, which is a larger version of the data center from Section 5.3. The boxes denote routers. Using terminology for fat tree networks [3], S[1–2] are spine routers, A[1–8] are aggregation routers (not related to BGP route aggregation), and T[1–8] are top-of-rack (ToR) routers. The spine routers connect to the Internet through neighbors N[1–2]. The ToR routers attach to a set of servers ("a rack") that host services with address prefixes P[1–8]. The intended policy for this network is as follows:

1. complete internal connectivity, *i.e.*, all routers should be able to reach each other;
2. services in Pods[1–2] should be accessible from outside;
3. prefixes for global service should be aggregated (as PG) when announced outside;
4. services in Pods[3–4] should not be externally accessible;
5. traffic paths should be valley-free (*e.g.*, a path through S1 should not go down through Ai and then back up through S2, for instance, creating an up-down-up path);

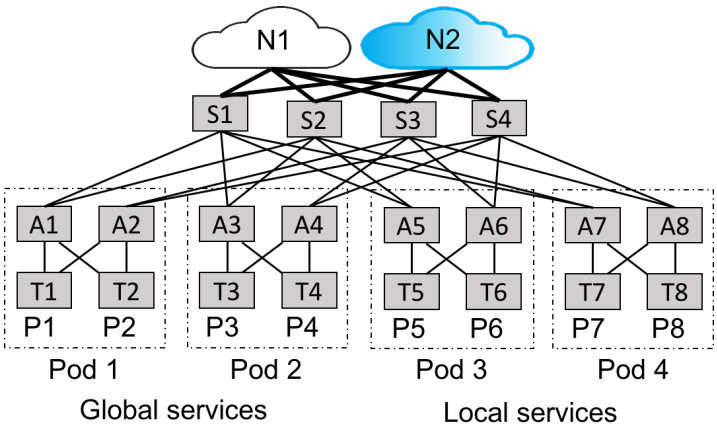


Figure 6.2: A modified version of the network in Figure 6.1.

```
%{for $n in neighbors do}%
interface $n.interface
   ip address $n.interfaceIP
!
%{end for}%

router bgp %{$bgpASN}%
   no synchronization
   bgp router-id %{$routerID}%
   bgp bestpath compare-routerid
%{for $n in neighbors do}%
   neighbor $n.IP remote-as $n. ASN
   neighbor $n.IP send-community both
   neighbor $n.IP route-map peer-in in
   %{if $n.isLocalPeer then}%
   neighbor $n.IP route-map local out
   %{else}%
   neighbor $n.IP route-map global out
   %{end if}%
%{end for}%
```

```
%{for $p in localPrefixes do}%
ip prefix-list localPL permit $p
%{end for}%
!
ip prefix-list other permit 0.0.0.0/0 le 32
!
route-map local permit 10
    match ip address prefix-list localPL
!
route-map local deny 20
    match ip address prefix-list other
!
route-map peer-in permit 10
    match as-path list WAN1
    set community additive 64512:3200
!
route-map peer-in permit 20
   match as-path list WAN2
   set local-preference 90
   set community additive 64512:3201
```

Figure 6.3: Idealized configuration template component for the data center spine.

6. prefer neighbor N1 over N2, *i.e.*, when both announce a prefix, use N1;
7. routers should not transit traffic between N1 and N2; and
8. no loss in connectivity after any single-link failure.

To correctly configure this policy, operators must generate configurations for each router, which implies ensuring, for instance, that all routing adjacencies are correctly configured (*e.g.*, T1's con-

figuration includes A1 as neighbor and vice-versa); the ToRs announce the correct prefixes for their services; all routers forward the prefix announcements that they should to each neighbor and not forward others (*e.g.*, the spines should forward prefixes for local services to internal neighbors but not to external neighbors); and the spines announce externally only the covering global prefix.

To configure this policy, an operator might adopt a template-based approach [61, 97]. Instead of authoring a configuration per router, operators author a template per role. A *role* is a specific function that is served by one or more routers. For example, the network in Figure 6.1 might have five roles: spine, global aggregator, global ToR, local aggregator, and local ToR. Figure 6.3 shows an example of what a small component of a template for the spine role in the two data centers might look like. The template has parameters for various aspects of the configuration (*e.g.*, neighbor list, local prefixes) and is compiled to low-level device configurations by instantiating the parameters using the network topology and a database of network information.

As described earlier, templates are hard to author and hard to validate. Worse, templates that work for one topology may not work for seemingly-inconsequential variations which may arise after the network evolves. Consider the network in Figure 6.2, which is similar to Figure 6.1; it has the same five roles, connected in a similar hierarchy. One might think that the same templates, with different database entries, can be used for both cases. However, if the templates are configured to disallow "valley" paths (per policy (5) above), they will work for Figure 6.1 but silently violate the fault tolerance policy (8) when used for Figure 6.2. Specifically, in Figure 6.2, an aggregation-based black hole will occur when the link S1–A1 fails; after this failure, S1 has no valley-free path to P[1–2] even though it will continue to get traffic for these prefixes as it announces the covering prefix PG (because it gets routes for P[3–4]). Such a black hole will not occur in Figure 6.1 because spine routers have two links to each pod.

When operators discover that an old template no longer works, they may consider changing it, which may cause a change to all devices that use it—an unacceptable disruption in many cases. As a result, operators may abandon the template entirely and revert to hand-crafting configuration

patches to accommodate the change. Such patches reintroduce the complexity and the risk of errors that templates were meant to prevent.

6.3 Topology Abstraction

Abstract topologies in Propane/AT define structural and role-based invariants that compactly describe all concrete networks that can emerge as the network evolves. Like the forall-exists ($\forall\exists-\mathsf{abstraction}$) abstraction for verification, they are encoded in the form of a graph homomorphism (*i.e.*, a mapping from concrete to abstract node). However to capture invariants about how the network might evolve, we allow these homomorphisms to be annotated with logical constraints about node and edge multiplicities. We designed the abstractions to be able to precisely capture real network topologies, while being amenable to fault-tolerance analysis in the abstract domain.

The topology abstractions consist of several concepts. The primary one is a role-based abstraction that allows an operator to map routers in the concrete network to roles in the abstract network. Figure 6.4 shows an example of this abstraction for both networks from Section 6.2. In the example, the concrete networks are abstracted into a new topology with 5 different roles: local ToR (TL), global ToR (TG), local aggregator (AL), global aggregator (AG), and spine (S).

More specifically, a network topology is a graph G = (V, E), which consists of a set of vertices V and a set of directed edges E: $V \times V$. A role-based abstraction is a graph homomorphism from G to an abstract graph G^A = (V^A,E^A). A graph homomorphism $f : V \rightarrow V^A$ (often written as $f : G \rightarrow G^A$) transforms a graph by mapping each node in the concrete graph to a node in the abstract graph such that, whenever $(u, v) \in E$, then $(f(u), f(v)) \in E^A$. The role-based abstraction therefore over-approximates the connectivity of the underlying concrete graphs.

On its own, this abstraction loses a lot of information about the concrete network's structure, making it difficult to reason precisely about fault-tolerance. For example, with this abstraction any spine router may or may not connect to any aggregator router. To capture concrete networks more precisely, we introduce additional concepts. The first is topology hierarchy, captured by P and Q,

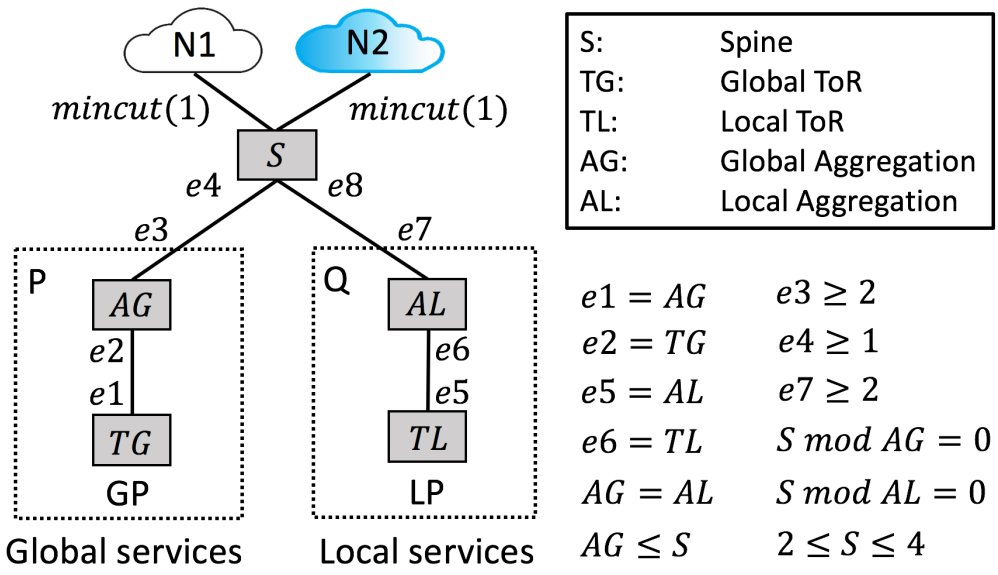


Figure 6.4: An abstraction for the network in Figure 6.1.

which indicate that nodes in the ToR and aggregator roles are grouped into pods. The second is node and edge *multiplicity*. Each edge (and node) is labeled with a symbolic variable (*e.g.*, $e1$) that denotes a constraint on the number of edges (and nodes) that may appear in any valid concrete network. Operators can capture concrete network invariants by adding constraints on the symbolic variables using logical formulas.

For example, in Figure 6.4, the first constraint ($e1 = AG$) states that, within any pod P, the number of outgoing edges from a node in the `TG` role (*i.e.*, $e1$) to a node in the `AG` role equals the number of nodes in the `AG` role. Similarly, the constraint ($e2 = TG$) states that the number of outgoing edges from a node in the `AG` role (*i.e.*, $e2$) to a node in the `TG` role equals the number of nodes in the `TG` role. Together these constraints capture the fact that, within any pod, the global aggregators and ToRs are in a full mesh. Furthermore, the constraints $AG = AL$ and $AG \leq S$ ensure that, within pods `P` and `Q`, the `AG` and `AL` roles have the same number of routers, which is less than or equal to the number of routers in the spine role `S`. The constraint $e3 \geq 2$ says that, in each pod, each aggregator node has at least 2 outgoing edges to nodes in the spine role. Symmetrically, the constraint $e4 \geq 1$ says that, for each pod `P`, each node in the spine role has at least one outgoing edge to a node in the AG role. Similar constraints appear for the local

aggregator role. The constraint $2 \leq S \leq 4$ makes explicit the possibility for growth, for example, by growing the network from Figure 6.1 to Figure 6.2. In general, we need not bound the number of spine routers to admit more concrete topologies, potentially at the expense of analysis precision. We also include the constraints $(S \mod AG) = 0$, and $(S \mod AL) = 0$ simply to show that constraints do not have to be in the form of inequalities. Operators can use logical formulas from any theory supported by modern SMT solvers.

A final concept is the $\mathrm{mincut}(1)$ constraint between the spine role `S` and `N[1--2]`. It says that any spine router has at least one path to any node in the neighbor `N1` (and `N2`) role. Such annotations are useful for a "one big switch" abstraction [25] in which a complex, unstructured network is represented as a single node. As another use case, an ISP backbone can be modeled by dividing the network into separate geographic regions with two roles per region—one for the border routers and another for the network core. Mincut annotations can describe the degree of fault tolerance both within regional cores and across regions.

The topology abstractions can also capture concrete topologies by using a one-to-one correspondence between abstract and concrete nodes/edges. This allows operators to define complex networks in which some (e.g., legacy) parts of the network cannot evolve while others can.

6.4 Policy Abstraction

Routing policies in Propane/AT are almost identical to those of Propane, but they differ in that they allow for predicates in the form of template variables. Let us see how to express the routing policy of the networks in Section 6.2 over the abstract topology. We can capture the basic routing behavior, constraints (1, 2, 6), as follows:

```
define Routing =
    {$GP  => end(TG)
     $LP  => end(TL)
     true => end(out) & exit(N1 >> N2)}
```

The second line introduces a prefix *template* variable `$GP`. Template variables represent multiple instances of a rule for different concrete prefixes that can be provided by an external source

(*e.g.*, a database). The line says that traffic for each global prefix associated with the variable should follow a path that ends at a destination router in the `TG` role (the particular router can be specified during concretization). The second line has a similar policy for local prefixes. The final rule matches all other IP prefix destinations and allows traffic to follow a path that leaves the data center, ending at some external role (**`out`**), through neighbors `N1` or `N2` with a preference for leaving through `N1`. Next, we can capture constraint (4) that traffic for local prefixes must stay within in the data center. The constraint is the same as in Propane:

```
define Local =
    {$LP => only(in)}
```

The constraint to prevent "valleys" (5) is written as:

```
define NoValley =
    {true => novalley({TG,TL},{AG,AL},{S})}
```

This policy applies to all traffic and prevents valley paths by adding the **`novalley`** constraint with arguments corresponding to each level in the data center. Constraint (7) to prevent transit traffic between neighbors is expressed as follows.

```
define Peer = {N1,N2}
define NoTransit =
    {true => !(enter(Peer) & exit(Peer))}
```

We define a `Peer` as `N1` or `N2` and disallow paths where traffic both enters and exits the data center through a peer.

Finally, we can combine these constraints expressed as follows.

```
Routing & Local & NoTransit &
NoValley & agg(GP_AGG, in -> out)
```

This policy is very similar to the one used for the data center in Chapter 5. In this case, GP_AGG is declared as a concrete prefix rather than a template because a single aggregate prefix will be used to summarize all less-specific prefixes. The complete routing policy is shown in Figure 6.5.

```
define Routing =
  {$GP  => end(TG)
   $LP  => end(TL)
   true => end(out) & exit(N1 >> N2)}

define Local =
   {$LP => only(in)}

define NoValley =
   {true => novalley({TG,TL},{AG,AL},{S})}

define Peer = {N1,N2}
define NoTransit =
       {true => !(enter(Peer) & exit(Peer))}

Routing & Local & NoTransit &
NoValley & agg(GP_AGG, in -> out)
```

Figure 6.5: Complete Propane policy for the abstract datacenter network.

Fault-Tolerance Policy: Operators may also specify how many link failures the network should be able to withstand before traffic experiences connectivity loss. Operators can set different tolerance levels for different pairs of abstract nodes. For instance, they may say that ToR to spine connectivity should be robust to 2 failures, *i.e.*, no ToR-spine pair should lose connectivity as long as the number of simultaneous link failures is 2 or fewer; and ToR-to-ToR connectivity should be robust to 1 failure.

6.5 Extending the PG for Abstraction

Figure 6.6 shows the product graph for the data center policy that applies to all external traffic: `true => exit(N1 >> N2)`. The first automaton represents the more preferred set of paths that exit through `N1` (`exit(N1)`) and the second automaton represents the less preferred set of paths that

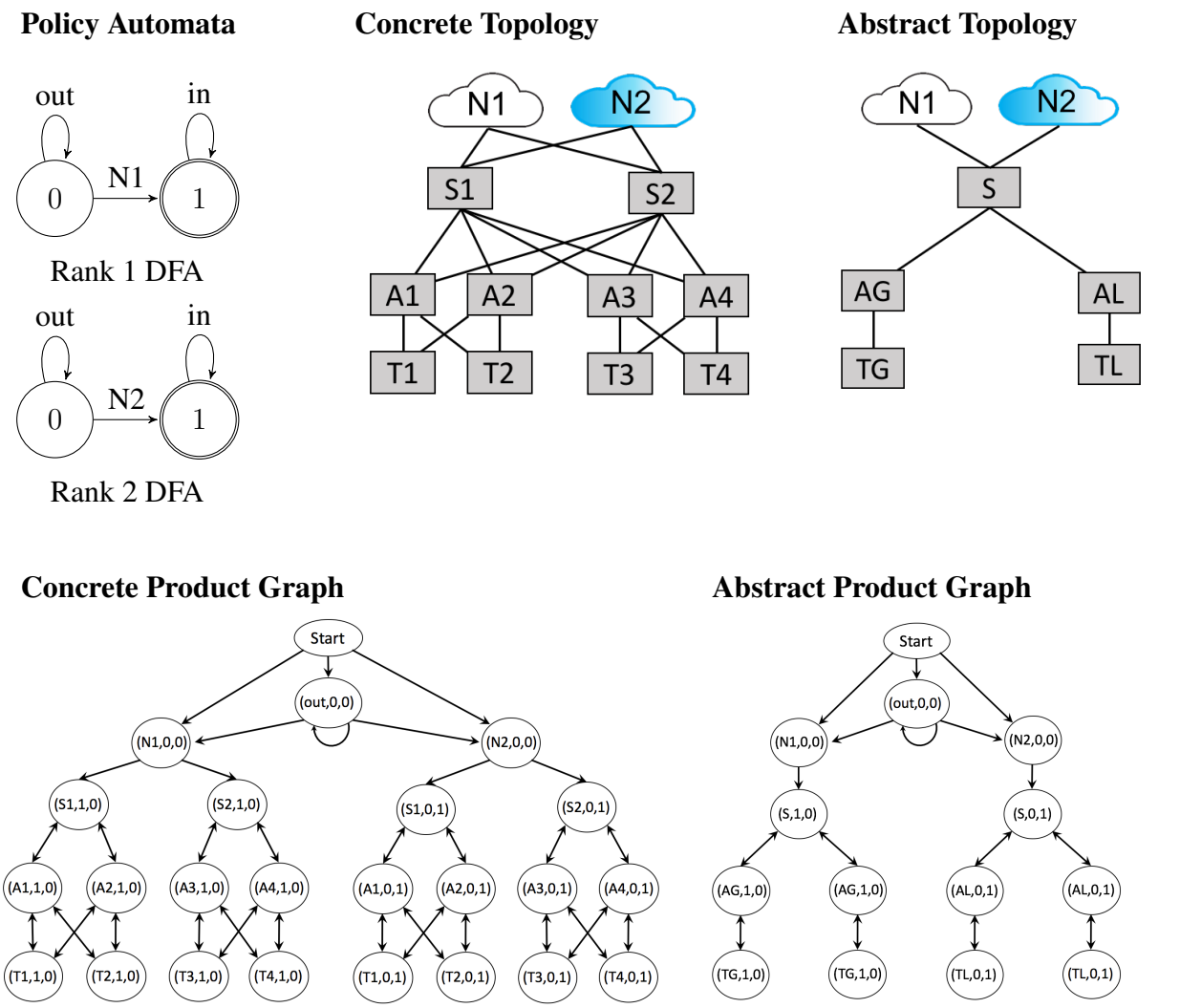


Figure 6.6: Product Graph construction for policy `true => exit(N1 >> N2)`.

leave through `N2` (`exit(N2)`). The PG is shown for both an instance of a simple concrete network matching the abstraction from Section 6.3 as well as for the abstract topology.

Interestingly, just as the concrete and abstract topologies are related, the concrete and abstract product graphs also have a very similar structure. In particular, we observe:

Lemma 6.5.1. *If we have a graph homomorphism* $f : G \to G^A$*, concrete product graph* $PG = (G', start, \text{rank})$ *and abstract product graph* $PG^A = (G'^A, start^A, \text{rank}^A)$*, then there is a homo-*

morphism $f_{pg} : G' \rightarrow G'^A$ *where:*

$$\begin{aligned} f_{pg}(start) &= start^A \\ f_{pg}((l, q_1, \ldots, q_n)) &= (f(l), q_1, \ldots, q_n) \end{aligned}$$

In other words, we can lift the topology homomorphism to relate the resulting product graphs. This fact ends up being important for synthesizing provably-correct templates. In particular, using this fact, we will show in Section 6.7 that our generation strategy commutes with template instantiation, meaning that we obtain the same results if we instantiate the abstraction early or if we defer the instantiation until after template generation.

6.6 Fault-tolerance Analysis

The possibility of network failures exacerbates the difficulty of constructing correct configurations. Link failures in networks occur frequently; it is not uncommon for a large network to experience dozens of failures in any given day [51]. However, existing tools [89, 14] reason about fault-tolerance only for concrete topologies. In contrast, Propane/AT provides a stronger guarantee: all possible concrete instantiations of an abstract topology satisfy the fault-tolerance policy.

We frame satisfying the fault-tolerance requirements as an analysis problem over the structure of the PG. In particular, we develop an analysis that uses information embedded in the abstract topology to infer bounds on the number of edge-disjoint paths between pairs of concrete nodes (much like the aggregation-safety analysis from Section 5.7). For each node in the abstract product graph, the idea is to infer facts learned about the number of edge-disjoint paths to groups of concrete routers corresponding to another abstract product graph node. More specifically, we maintain fact of the form:

$$L_{X_1}, \ldots, L_{X_n}(j, k)$$

where each label $L \in \{S, A\}$ is either S, which stands for "some" or is A, which stands for "all". There is one label for each pod in the abstraction pod hierarchy under which the abstract

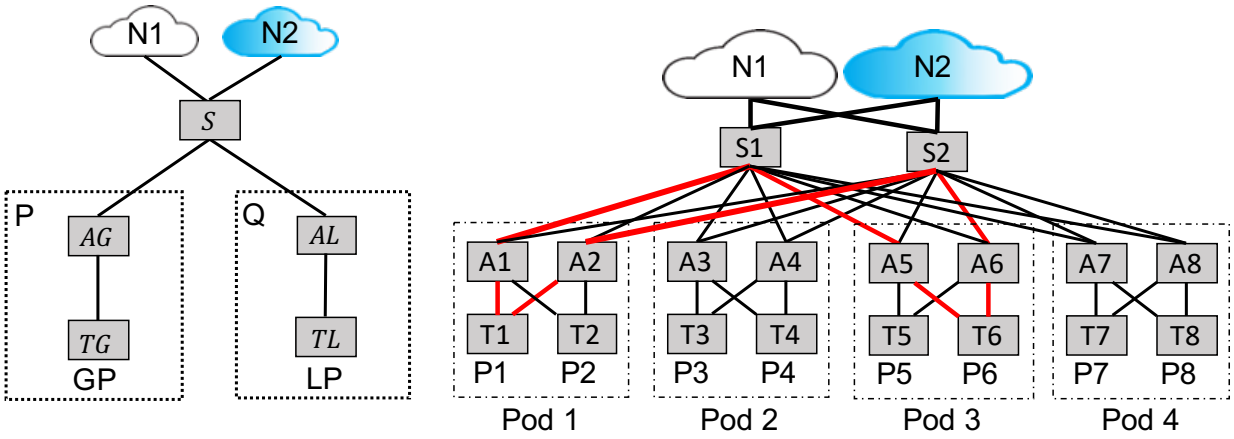


Figure 6.7: Example of a sound inference for the data center running example.

node appears. For a given node, L_{X_1} corresponds to the outermost pod, $L_{X_{n-1}}$ corresponds to the innermost pod, and L_{X_n} to the node itself. Semantically, $L_{X_1}, \ldots, L_{X_n}(j, k)$ means that starting from some concrete source node, for some/all pods $X_1, \ldots, X_{n-1}$ and for some/all groups of nodes in the role X_n of size j, there are k paths to each such that all $j * k$ paths are edge-disjoint.

For example, an inference of the form $A_Q A_{TL}(2, 3)$ states that, from the given source location, for *all* pods Q, and *all* groups of 2 nodes in the TL role in pod Q, there are 6 disjoint paths to the group—3 for each of the 2 nodes. Consider the running data center example in Figure 6.7. For this example, the inference $A_Q A_{TL}(1, 2)$ is a sound inference, since each node in the TL role in any of the two pods on the right (Q) has at least two disjoint paths from any source node in the TG role. The figure shows two such disjoint paths between two particular nodes, but symmetric paths exist for other nodes.

6.6.1 Inference Rules

Figure 6.8 displays the collection of rules used to infer facts about disjoint paths. Each rule is read from bottom to top. The label on the bottom left is a known fact. We use L to represent a rule that is parametric over the label (S or A). Labels on other nodes correspond to facts learned

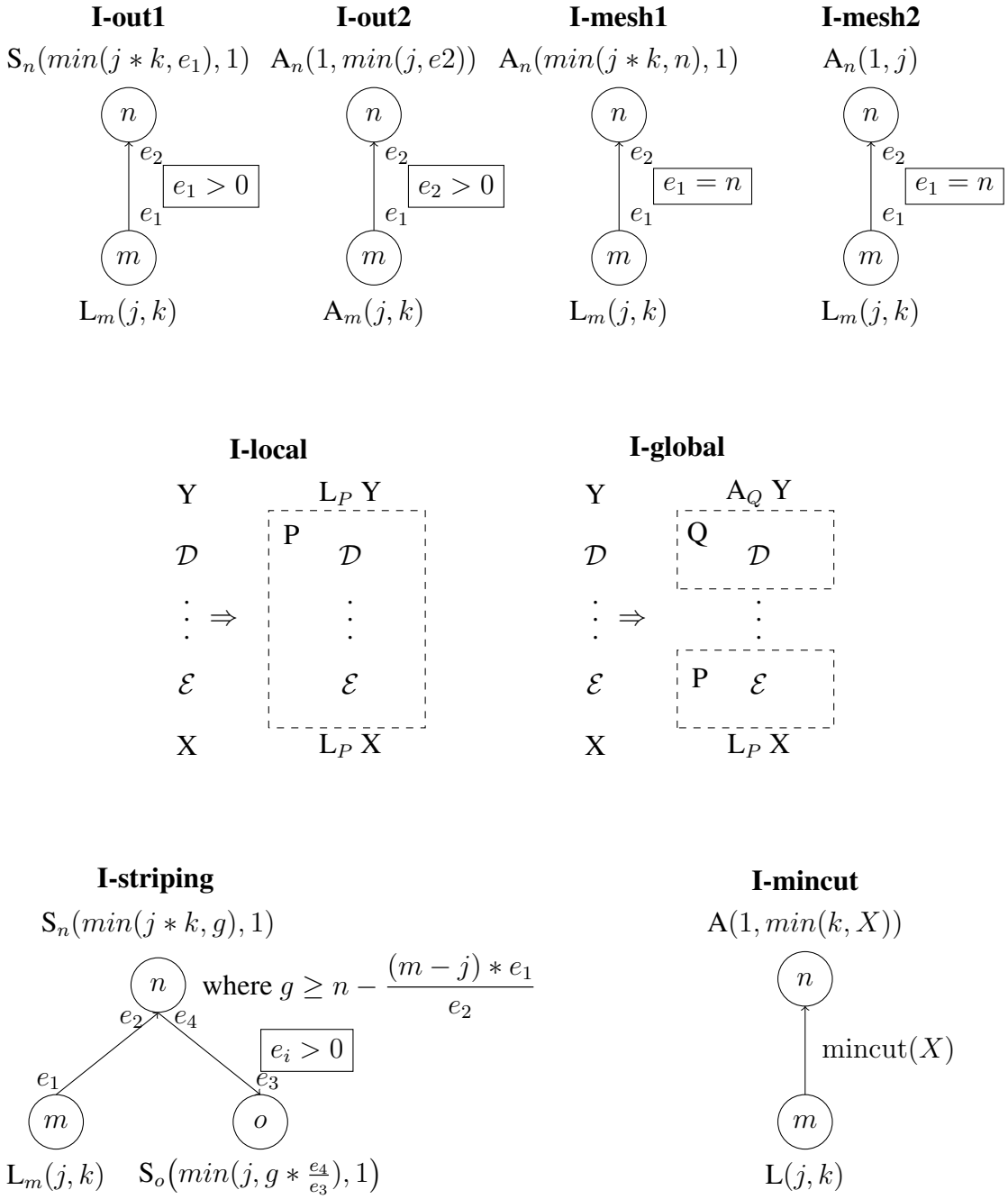


Figure 6.8: Abstract k-disjoint path analysis inference rules.

after applying the rule. The box shows the conditions that must be valid, given the abstraction constraints, for the rule to apply.

Some of the inference rules (*e.g.*, I-out2 and I-mesh2) try to learn about the largest number of disjoint paths to any single node in an abstract role, while others (*e.g.*, I-out1 and I-mesh1) try to

learn about the largest reachable group in a particular role with at least one disjoint path to each node in that group. Both kinds of rules are useful.

The first rule, l-out1 applies to a learned fact of the form $L_m(j,k)$ where the number of outgoing edges from any concrete node in the m role is greater than 0. In the worst case, the largest group of concrete nodes we could hope to reach at the n role would be e_1 since all j nodes at the bottom may have outgoing edges to the same concrete nodes at the top. Furthermore, the total number of disjoint paths to the j nodes at the bottom is equal to $j * k$. Since extending the existing disjoint paths with disjoint edges keeps the paths disjoint, and since we cannot exceed the current number of disjoint paths to the concrete nodes in role m on the bottom, the largest reachable group for the role n on the top will be $\min(j * k, e_1)$. We conservatively use 1 for the number of disjoint paths to each node in role n, since when n is very large, all reachable nodes in role m might only have a single edge to completely different nodes in n.

Consider rule l-out2, and consider any node in the role n. There are e_2 incoming edges to that node. Due to the fact that $A_m(j,k)$, we know that at least j of those e_2 edges are connected to nodes with disjoint paths from the origin. Hence we infer $A_n(1, \min(j, e_2))$.

The two rules l-mesh1 and l-mesh2 handle the case where there is a full mesh between the two roles. This happens when the number of outgoing edges (e_1) from nodes in role m equals the number of nodes on top (n). l-mesh1 says that we can find disjoint paths to each node in the top role restricted to the number of disjoint paths we started with. l-mesh2 uses the fact that each node in the top role is connected to each node in the bottom role to infer that there can be j disjoint paths to any single node in the top role.

The annotation $\mathrm{mincut}(X)$ appearing on an edge is an assertion about the fault tolerance between nodes in two different roles. The rule l-mincut uses such assertions. To each node in role n, from a node in role m, we can construct at least the minimum of X and k disjoint paths.

The rule l-striping is the most complicated case. It starts with the invariant $L_m(j,k)$ at role m and can be applied if each edge multiplicity $e_i > 0$ is valid given the constraints. The first inference for role n tries to find the largest reachable group with disjoint paths to each. The idea is similar

to the rule l-out1, but is able to use the fact that $e_2 > 0$ to learn more about the structure of the concrete topology. In particular it uses the following inequality, where g represents the size of the group for the role n:

$$(m - j) * e_1 \geq (n - g) * e_2$$

The remaining nodes $(m - j)$ that are not part of the reachable group in the bottom role, each have e_1 outgoing edges and must be able to at least "fill" the incoming edges for the remaining nodes not in the reachable group at the top $(n - g)$, which each have e_2 incoming edges. Solving the inequality gives the lower bound for g used in Figure 6.8.

The second part of the rule uses a similar idea to reason about the overlap between roles m and o with respect to role n. This rule is particularly useful for data center topologies where routers in one tier of the data center often have a uniform striping pattern with another tier.

Finally, rules l-local and l-global reason across pod hierarchies. l-local says that if there is an inference from X to Y, then the derivation can be used inside pod P by leaving the P-label unchanged. l-global says that when the edge goes across pods, we can infer the fact for all pods Q since the multiplicities apply uniformly for each pod.

6.6.2 Inference Algorithm

The abstract disjoint path analysis starts from a fixed source location src and repeatedly tries to apply every inference rule from Figure 6.8 until it reaches a fixed point. The algorithm applies an inference rule when the rule's condition is valid given the abstract topology constraints. Because the inference rules may continue to yield larger and larger symbolic expressions, we make the following observation to ensure termination: for any invariant learned of the form $L(j, k)$, it is sound to instead infer $L(j', k')$ if $j' \leq j$ and $k' \leq k$. Therefore, for each inference $L(j, k)$ we minimize the value of the symbolic expressions for j and k subject to the topology constraints using the optimizing SMT solver νZ [21].

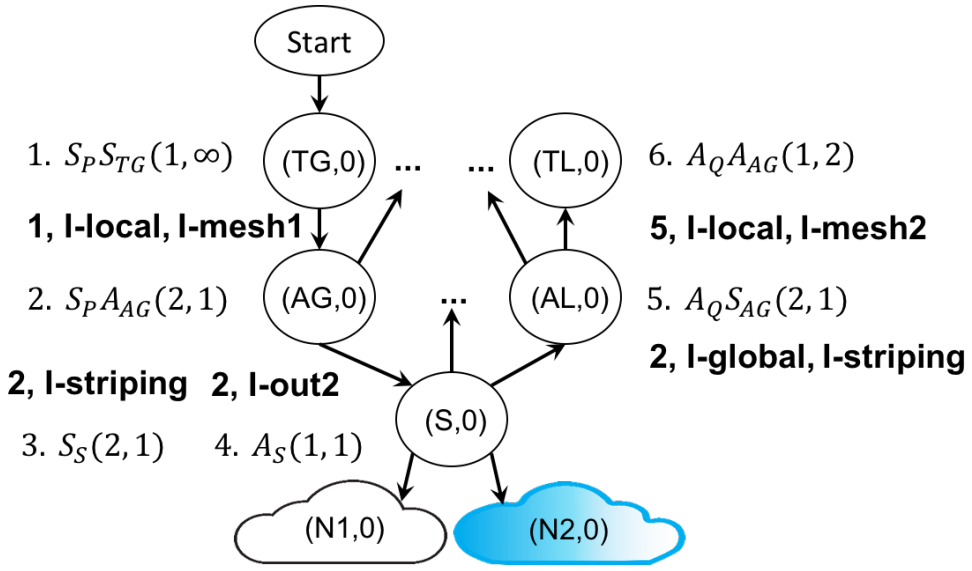


Figure 6.9: Abstract disjoint path analysis for global prefixes.

At a higher level, what is happening is that each inference rule is attempting to learn the maximum fault tolerance information possible as a function of the symbolic inputs. The νZ [21] solver will then minimize this maximum by accounting for all possible topologies that meet the abstraction. Facts learned with $j = 0$ or $k = 0$ are discarded. Recall the policy for global prefixes in the data center.

```
$GP => end(TG) ∩
       novalley({TG,TL},{AG,AL},{S}) ∩
       !(enter(Peer) & exit(Peer))
```

Figure 6.9 shows part of the abstract PG representation for this routing policy. The inference algorithm starts from the node $(TG, 0, 0)$ with the initial fact $S_P S_{TG}(1, \infty)$ (*i.e.*, no restriction on the number of disjoint paths initially). The first step applies each inference rule to this initial fact. The algorithm uses rules l-mesh1 and l-local to reason about connectivity within a single pod for the TG and AG roles. It makes a call to νZ to minimize the expression $\min(\infty, AG)$, which results in 2. Therefore, the algorithm learns a new invariant of the form $S_P A_{AG}(2, 1)$ for node $(AG, 0)$ to indicate that in some pod P, any group of 2 nodes is reachable. The algorithm will then eventually apply l-out2 to learn that any single spine node is reachable at $(S, 0)$. It will also apply l-striping to

determine that there is some group of at least 2 spine routers reachable at $(S, 0)$ and that there is some group of at least 2 nodes reachable in the AL role in state $(AL, 0)$.

Note that, because each inference rule only applies to directed edges in the PG, the algorithm cannot make any inferences about connectivity from the AL role to the S role since there is no directed edge from AL to S. This restriction ensures that the analysis remains policy-sensitive.

The next step is to use l-mesh2 together with l-local to infer that any single node in the TL role for any pod Q is reachable via at least 2 disjoint paths. This process will continue until a fixed point is reached.

The algorithm could infer that there is at least 1 disjoint path to any spine router, and at least 2 disjoint paths to any TL router. In this case, the analysis is precise. There exists a concrete network, namely the data center from Figure 5.2, where a single failure can disconnect a global ToR from a spine router due to the valley-free constraint.

6.7 Template Generation

The translation from the PG representation to per-device templates that run the distributed BGP protocol is the same as the translation used for Propane. However, rather than producing one configuration per device, the compiler will now produce one configuration per abstract device. Intuitively, each abstract configuration can be viewed as a template for all concrete devices that map to that abstract device.

Figure 6.10 (left) shows part of the generated mBGP configuration for spine routers for both the concrete and abstract policies. For brevity, we use the symbol $(*)$ to denote the set of all neighbors and omit tags when irrelevant. For prefix `true`, the spine routers will match advertisements from peer `N1` and `N2`. The match for `N1` is preferred since it has a higher BGP local-preference attribute (110). If an advertisement from `N1` is chosen, the spine attaches the community tag `(1,0)` before sending the route to all its peers $(*)$. If an advertisement is only available from the backup `N2`, then it attaches the tag `(0,1)` instead. The template configuration matches any global prefix `$GP`

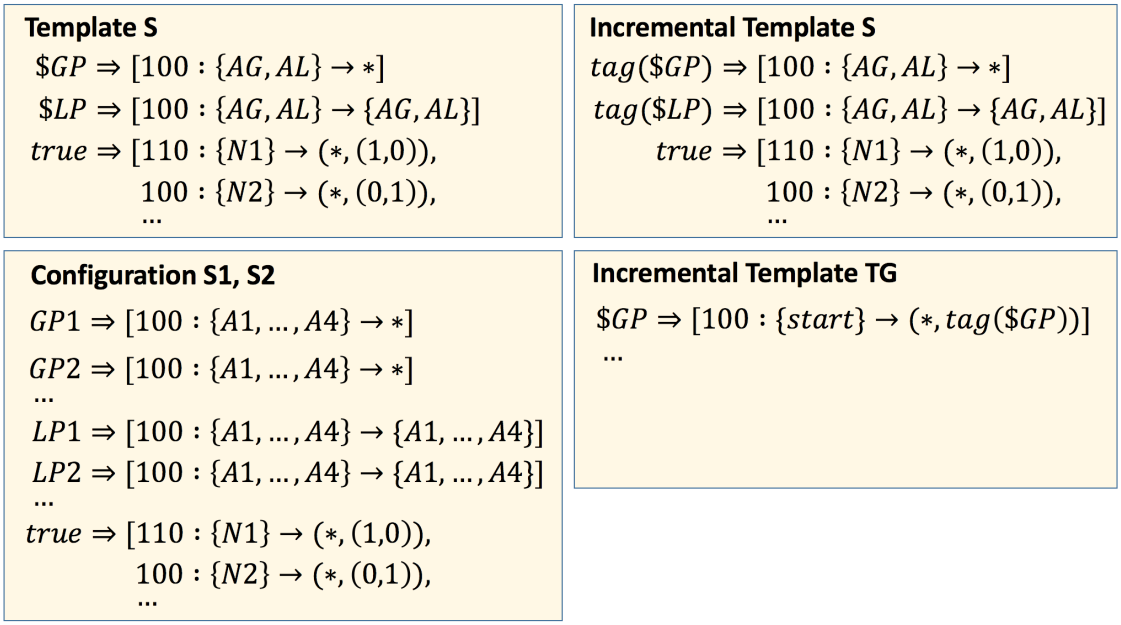


Figure 6.10: Spine template, concrete configurations, and evolution-friendly templates.

from any internal peer and re-advertises the route to all its peers. For any local prefix, it will allow an advertisement from any internal peer, and re-advertise the route to only other internal peers. The concrete configurations for S1 and S2 obtained from compilation for the concrete PG from Figure 6.6 have a similar structure for each local and global prefix where local routes are reflected downward, while global routes are advertised to all peers.

6.8 Concretization

One can observe that the inferred local preferences for concrete and abstract devices are identical. This leads us to prove the following relationship between the concrete and abstract product graphs:

Lemma 6.8.1. *$m \leq n$ in the concrete PG iff $f_{pg}(m) \leq f_{pg}(n)$ in the abstract PG.*

Lemma 6.8.1 tells us that inferring preferences for the abstract PG before template instantiation is equivalent to inferring preferences for an already-instantiated concrete PG. The similar structure between the spine template and concrete configurations is not a coincidence. We formalize this observation by defining two concretization functions in Figure 6.11 (con). One concretization

Propane/AT Concretization

$$
\begin{array}{lcl}
\text{con}(\emptyset, \Gamma, f) & = & \emptyset \\
\text{con}(l, \Gamma, f) & = & \Sigma\, f^{-1}(l) \\
\text{con}(r_1 \cup r_2, \Gamma, f) & = & \text{con}(r_1, \Gamma, f) \cup \text{con}(r_2, \Gamma, f) \\
\text{con}(r_1 \cap r_2, \Gamma, f) & = & \text{con}(r_1, \Gamma, f) \cap \text{con}(r_2, \Gamma, f) \\
\text{con}(!r, \Gamma, f) & = & !\text{con}(r, \Gamma, f) \\
\text{con}(r^*, \Gamma, f) & = & \text{con}(r, \Gamma, f)^* \\
\text{con}(pfx \Rightarrow r_1,\ \ldots\ , r_k, \Gamma, f) & = & pfx \Rightarrow \text{con}(r_1, \Gamma, f),\ \ldots\ , \text{con}(r_k, \Gamma, f) \\
\text{con}(\$x \Rightarrow r_1,\ \ldots\ , r_k, \Gamma, f) & = & [\ pfx \Rightarrow \text{con}(r_1, \Gamma, f) \cap \text{end}(l),\ \ldots\ , \\
 & & \qquad \text{con}(r_k, \Gamma, f) \cap \text{end}(l) \mid (pfx, l) \in \Gamma(x)\ ] \\
\text{con}(p_1, \ldots, p_n, \Gamma, f) & = & \text{con}(p_1, \Gamma, f), \ldots, \text{con}(p_n, \Gamma, f)
\end{array}
$$

mBGP Concretization

$$
\begin{array}{lcl}
\text{con}(l_i \rightarrow rc_i, \Gamma, f, G) & = & append_i\ [\ \ell \rightarrow con(rc_i, \ell, \Gamma, f, G) \mid \ell \in f^{-1}(l_i)\ ] \\
\text{con}(t_i \rightarrow pc_i, \ell, \Gamma, f, G) & = & \text{con}(t_1 \rightarrow pc_1, \ell, \Gamma, f, G), \ldots, \text{con}(t_k \rightarrow pc_k, \ell, \Gamma, f, G) \\
\text{con}(pfx \rightarrow pc, \ell, \Gamma, f, G) & = & pfx \rightarrow \text{con}(pc, \text{true}, \ell, \Gamma, f, G) \\
\text{con}(\$x \rightarrow pc, \ell, \Gamma, f, G) & = & [\ pfx \rightarrow \text{con}(pc, l = \ell, \ell, \Gamma, f, G) \mid (pfx, l) \in \Gamma(x)\ ] \\
\text{con}(ma_i, o, \ell, \Gamma, f, G) & = & \text{con}(ma_1, o, \ell, \Gamma, f, G), \ldots, \text{con}(ma_k, o, \ell, \Gamma, f, G) \\
\text{con}(d : (n_1, c_1) \rightarrow & = & \textbf{if } n_1 = \{\text{start}\} \text{ and } o = \text{false } \textbf{then } \bullet \\
\quad (n_2, c_2), o, \ell, \Gamma, f, G) & & \textbf{else } d : (\text{con}(n_1, \ell, \Gamma, f, G), c_1) \rightarrow (\text{con}(n_2, \ell, \Gamma, f, G), c_2) \\
\text{con}(\{l_1, \ldots, l_k\}, \ell, \Gamma, f, G) & = & \bigcup_i\ \{x \mid x \in f^{-1}(l_i),\ (x, \ell) \in G.E\}
\end{array}
$$

Figure 6.11: Propane/AT policy and mBGP concretization functions.

function is for Propane/AT policies and another is for mBGP policies. Concretization takes a context $\Gamma : Var \rightarrow 2^{Prefix \times V}$ that maps each template variable to a set of pairs of a concrete prefix and topology location where the prefix is owned. Both concretization functions traverse the policy and substitute instances of a topology location l in the template policy with the set of all concrete locations that map to l, given by the inverse homomorphism $f^{-1}(l) = \{l' \mid f(l') = l\}$. Additionally, whenever a pair $(pfx, l) \in \Gamma(x)$, then a new entry is added to the concretized policy where pfx replaces $\$x$ and adds the constraint that traffic ends at l (`end(l)`). For example, the spine template in Figure 6.10, is obtained by substituting $\{A1, A2\}$ for AG and $\{A3, A4\}$ for AL and by also replacing the entry for $GP with entries for GP1 and GP2 given by the context.

We prove that the compilation and concretization functions commute (the full proof can be found in Appendix Section A.2):

Theorem 6.8.2. *For any context* Γ*, topologies* G *and* G^A*, homomorphism* $f : G \to G^A$*, and policy* pol*:*

$$\mathit{con}(\mathit{compile}(pol, G^A), \Gamma, f, G) = \mathit{compile}(\mathit{con}(pol, \Gamma, f), G)$$

This is a powerful result, because it means that the order of concretization does not matter. If we had first concretized the policy and then performed synthesis over a concrete policy, we would get the exact same configurations if we first perform synthesis over the abstract topology to get templates, and only then concretize the templates. Since synthesis is relatively expensive compared to concretization, it pays to perform synthesis over the smaller abstract network, and only then concretize the resulting templates.

6.9 Incrementality

Suppose an operator wants to expand the concrete data center from Figure 6.1 by adding an additional ToR router to the TG role. Per the network routing policy, the new router will advertise any owned prefixes provided by looking up `$GP` in Γ. Because the new topology matches the abstraction, the compiled templates will remain the same. However, in the spine configurations, the match on the global prefix template variable `$GP` must be expanded when concretizing the template to include the new prefixes added by the ToR. Hence, this small change to the topology results in a change to every single spine configuration.

More generally, each configuration template depends on two things: the routing policy and the abstract topology. If the policy remains fixed and a change to the concrete topology preserves the topology abstraction, then the generated templates will not change. Further, each template has policy only in terms of its immediate neighbors. Because abstract neighbors are substituted for concrete neighbors during concretization, it would seem as though the generated configurations will also only depend on their concrete neighbors. However, prefix template variables allow for the possibility of introducing new prefixes in the context Γ after a change. For example, when

adding a new ToR router with its own unique prefix, the spine configurations would need to know about this new prefix. In fact, the only way in which the concrete configurations can depend on anything non-local is when instantiating prefix template variables.

To prevent the non-local changes induced by template variables, we modify compilation in the following ways. First, we associate a new unique community tag for each template variable (*e.g.*, `$GP`), and add this tag where the route is originated (*e.g.*, role TG). Then, template variable tests elsewhere in the policy are replaced with a new test on this tag. Finally, during template instantiation the tags are left unmodified. Figure 6.10 (right) shows the spine and ToR templates after this transformation. Routers in the TG role will originate ($\{\mathtt{start}\}$) global prefixes and tag them with a unique tag, while routers in the spine role match the tag.

6.10 Implementation

Propane/AT is implemented in roughly 3000 additional lines of F# code as an extension to the original Propane compiler. The Propane/AT compiler generates configurations for Cisco and Quagga [85] routers. The fault-tolerance analysis uses νZ [21] to both test validity and minimize variables subject to the topology constraints. Since the analysis typically calls the SMT solver many times with relatively small optimization problems, we use a timeout of 200ms.

Although the disjoint path analysis takes place over the PG, each application of the inference rules from Figure 6.8 depends on the topology locations, but not the automata states, and can be reused across multiple PG nodes with the same topology location. Therefore, we lazily apply the rules and cache the satisfiability and minimization calls to νZ after their first use. Furthermore, the cached results are shared across different prefixes, each of which may have a unique PG representation.

	Fixed	Reachability		K-paths	
		Some Pairs	All Pairs	Some Pairs	All Pairs
Tree-based topologies, valley-free routing					
Fat tree [3]	–	✓	✓	✓	✓
Facebook [7]	–	✓	✓	✓	✓
F10 [76]	–	✓	✓	✓	✓
VL2 [54]	–	✓	✓	✓	✓
All topologies, shortest-path routing					
Fat tree [3]	–	✓	✓	✓	C
Facebook [7]	–	✓	✓	✓	✓
F10 [76]	–	✓	✓	✓	C
VL2 [54]	–	✓	✓	✓	C
BCube [59]	k	✓	✓	C	C
DCell [60]	k	✓	✓	C	C
Butterfly [70]	n	✓	✓	✓	✓
Hypercube	N	✓	✓	✓	✓
HyperX [2]	L	✓	✓	✓	✓

Figure 6.12: Expressiveness and precision of Propane/AT.

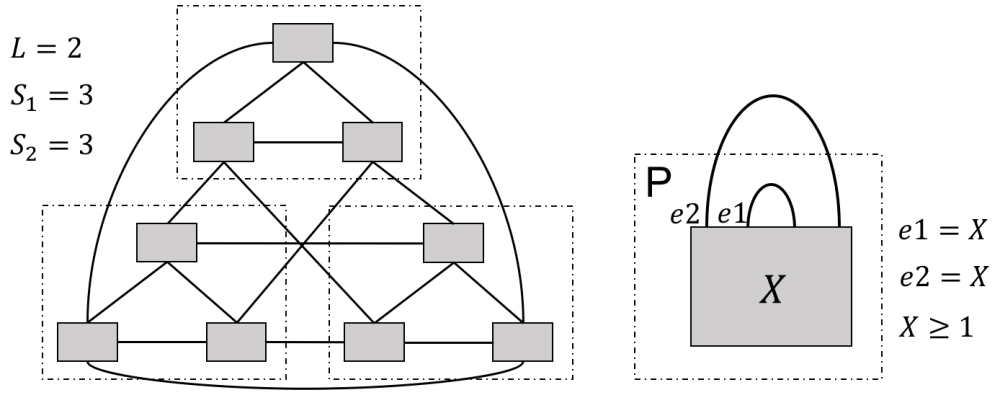
6.11 Evaluation

6.11.1 Expressiveness and Precision

We evaluate the expressiveness of Propane/AT's topology abstractions and the precision of its fault-tolerance analysis on a range of network topologies found in production networks and in the networking literature. We characterize expressiveness by checking if the abstractions allow the topologies to evolve arbitrarily or certain aspects must be fixed (*i.e.*, cannot be symbolic). We measure precision by checking if we find a tight lower-bound on fault-tolerance (*i.e.*, there is a concrete network with that degree of fault-tolerance).

The top part of Figure 6.12 shows the results for common data center networks: tree-based topologies coupled with valley-free routing. We consider four variants of tree topologies: a standard fat tree [3], the Facebook fat tree [7], the F10 fault-tolerant fat tree [76], and VL2 [54]. These variants differ in the number of tiers and the connectivity pattern between roles. For each, we use a tiered abstraction similar to that in our example (Section 6.3) and parameterize over the number of

HyperX



BCube

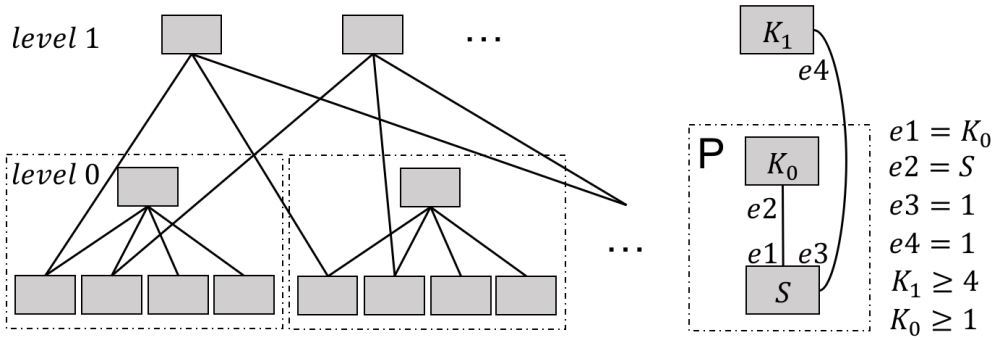


Figure 6.13: Example abstractions for HyperX and BCube.

pods, which can be scaled for expansion. We report precision of both analyzing reachability and disjoint paths, and we report if Propane/AT is precise for all pairs of abstract nodes or only some of them. We record a check when the analysis is precise and a C when the analysis is conservative.

Our results are encouraging for these settings. Our abstractions are perfectly expressive for tree-based topologies—we did not have to fix any aspect of their structure—and the analysis is precise in all cases. To stress our abstractions and analysis, we consider several additional topologies that appear in the literature.

Recursive Topologies: These topologies include BCube [59] and DCell [60]. Each topology includes a recursion depth parameter (k), which we fixed while abstracting them. For a recursive topology with depth k, we model it as an abstract topology consisting of a pod to represent all depth $k-1$ subcomponents. This allows for safe expansion within a subcomponent, but does not

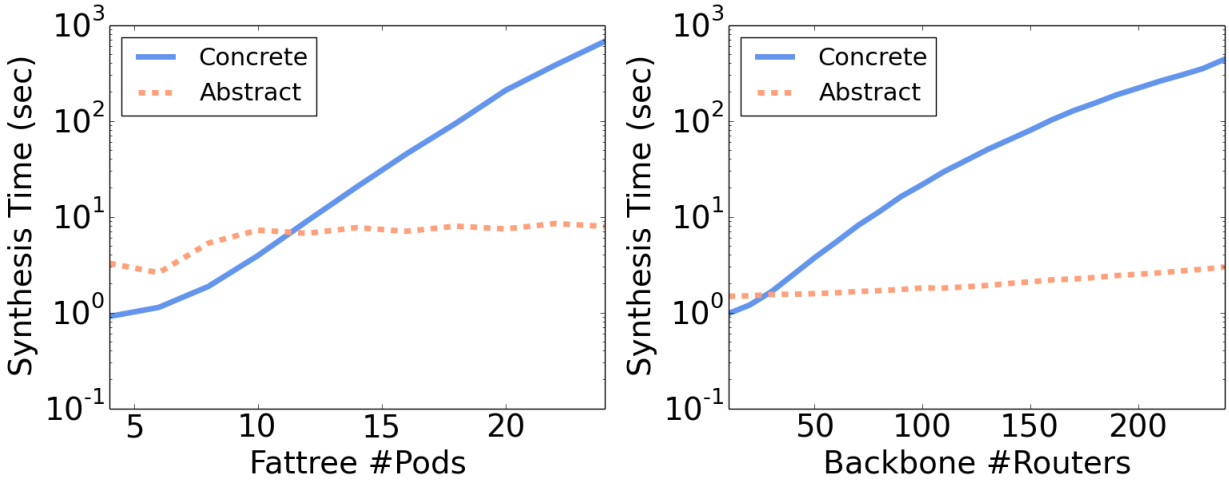
allow changing the recursion depth dynamically. For BCube, we model each tier of the data center as a separate role. Figure 6.13 shows an example of a BCube abstraction for $k = 1$.

Hypercube Topologies: Hypercube variants can be used as an alternative to Clos-style topologies for networks with port density routers. The HyperX [2] topology generalizes the hypercube and butterfly topologies and includes parameters L for the lattice dimension of the network, and S_i for the node multiplicity of each dimension i. For a fixed number of dimensions L, we abstract each full mesh of S_L nodes into its own abstract node. Nodes in dimension S_{x-1} are abstracted using pods of abstract nodes from dimension S_x. Figure 6.13 shows an example for $L = 2$.

Results: The bottom part of Figure 6.12 shows the results for all types of topologies with shortest-path routing. (Valley-free routing is not meaningful for non-tree-based topologies.) For all tree-based topologies, the analysis is precise for reachability, but for three of them, it does not compute a tight bound for disjoint paths for all router pairs. Specifically, it underestimates ToR-to-spine paths; it fails to account for some circuitous paths that traverse another spine because it could not disambiguate two concrete spines that map to the same abstract role. For instance, for the fat tree topology [3], it only finds 1 path between any ToR and any Spine when there should always be at least two. However, in this case the analysis computes the correct worst case connectivity between any source ToR and any other destination aggregation or ToR router. A similar pattern occurs with other tree-based topologies. For both recursive topologies, the analysis can only accurately determine reachability.

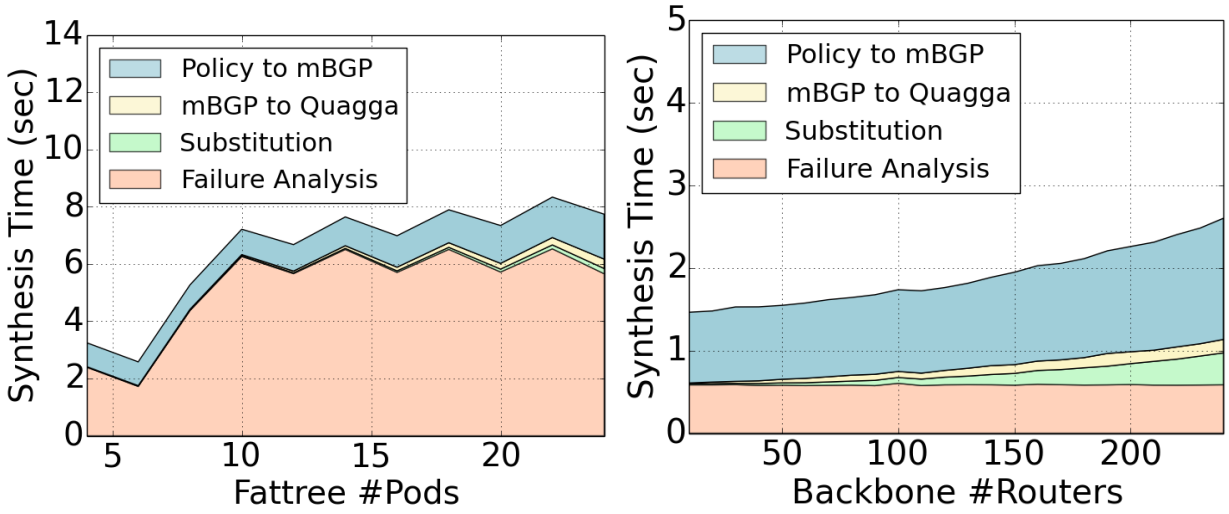
6.11.2 Synthesis time

We evaluate generation time in Propane/AT both with and without abstraction using routing policy for backbone and data center networks inspired by configurations obtained from a large cloud provider. For both types networks, we fix the routing policy and scale the size of the topology.



(a) Data center **(b)** Backbone

Figure 6.14: Concrete vs. Abstract Synthesis Time.



(a) Data center **(b)** Backbone

Figure 6.15: Abstract Synthesis Time by Phase.

Topologies: Routers in the data centers run BGP using unique AS numbers and connect to multiple external neighbors. The routers aggregate some prefix blocks when announcing them to external neighbors, and keep some prefixes internal. The data center prefers that traffic leave through certain neighbors over others and should not transit traffic between neighbors. The policy also prevents routers from using external neighbors to reach "private" destinations (i.e., those in the IP address space reserved for private use). We use a fat tree [3] and scale it by increasing the number

of pods. The abstract topology uses one abstract node for each tier with additional nodes for local and global ToRs.

The backbone policy classifies neighbors into several categories based on commercial relationship [47] and prefers paths through them in order. Like the data center, it blocks private destinations from neighbors, drops transit traffic between certain pairs of neighbors, and aggregates internal prefixes at the network border. We scale the backbone network from 10 to 240 routers. We split it into two parts: border routers that connect to external neighbors and an internal core. We use one abstract node for the border routers and one for the network core with mincut annotations both within the core (*i.e.*, with a self-loop) and between the core and border roles. For neighbors, there is one abstract role per commercial category.

Results: Figure 6.14 shows total configuration generation time for Propane/AT vs the concrete network synthesis tool Propane. All experiments were run on an 8 core, 2.4 GHz Intel i7 processor machine running Mac with 16GB of Ram.

For both networks, the abstract synthesis is slightly slower than concrete synthesis for small topologies due to the overhead of the fault-tolerance analysis. However, as the topology size increases, abstract synthesis becomes orders of magnitude faster. In all cases for both networks, it takes less than 10 seconds to complete.

Figure 6.15 shows the relative time taken by each phase of Propane/AT. The fault-tolerance analysis takes the most time, but that does not depend on the number of concrete nodes in the network, and thus is largely a fixed cost. In particular, the number of calls to νZ remains constant across topology size. The seesaw behavior for the data center networks results from differences in time taken by νZ to minimize similar constraints with different values.

6.11.3 Incrementality

Propane/AT's compilation strategy guarantees that network evolution requires configuration changes only for nodes that acquire or lose a neighbor. We experimentally confirmed that our

implementation provides this guarantee. For the networks we studied above, we made a range of changes, including adding and removing routers and pods and changing prefixes that routers originate. In each case, we found the guarantee to hold. In contrast, all router configurations were modified with Propane because it heavily uses prefix lists which are sensitive to such changes. While Propane may be made friendlier to network evolution, its fundamental limitation will remain because it does not understand roles and the network's structure that Propane/AT leverages.

6.12 Summary

To recap, in this section we once again investigated how to take advantage of network symmetry through abstraction. While Chapter 4 focused on abstraction for the purpose of verification, in the chapter we explored abstractions for synthesis. Although many of the high-level ideas are similar (*e.g.*, the notion of a $\forall\exists-$abstraction is useful in both contexts), there are many technical differences between the two approaches. However, as with verification, the use of abstraction is often very effective for speeding up network synthesis on large networks. When used in the context of synthesis, abstraction provides additional benefits such as the ability to generate correct parameterized networks with guaranteed fault-tolerance and incremental properties.

Chapter 7

Conclusion

Reliable communication over networks depends on routing, the process through which devices learn how to forward traffic to different destinations. Standard routing protocols are highly flexible, allowing network operators to achieve a variety of economic-, performance-, and robustness-related objectives. However, configuring routing protocols to achieve such objectives while ensuring end-to-end network correctness remains a challenging problem, made evident by the large number of configuration-related bugs and outages that occur in many production networks [6, 37, 53, 64, 87, 88, 90, 91, 95].

This dissertation has presented two complementary approaches to proactively addressing the problem of configuration complexity.

Verification: The first approach is formal verification to check that configurations conform to a high-level specification of the desired end-to-end network behavior. Formal verification is a powerful tool that can guarantee correctness of existing configurations for all possible data planes that can emerge from the network control plane. To make verification a reality for networks, we first developed a formal model of the network control plane in the form of the Stable Routing Problem (SRP) from Chapter 3. Given an SRP model of a network, we demonstrated how to translate this model into a collection of SMT constraints that directly characterize all possible stable data planes that can emerge from the control plane. Verification is made possible then by

leveraging SMT solvers to check there is no stable solution to the SRP where "bad behavior" is possible.

To scale verification to large networks, we considered two different approaches. The first was slicing and hoisting optimizations to the encodings into SMT constraints. By taking advantage of domain-specific knowledge, we could improve the performance of the SMT solver by several orders of magnitude. The second was through network *abstraction*. By levering symmetries that exist in configurations, we demonstrated how to take a large SRP as input and produce a smaller SRP with a provably equivalent space of solutions. By performing verification directly on the smaller SRP, verification can be sped up by several more orders of magnitude in many cases.

Synthesis: The second approach this dissertation explored is configuration synthesis. Rather than checking the correctness of configurations against a high-level specification, synthesis aims to generate correct configurations directly from the end-to-end specification. To make synthesis possible, we first developed a language for defining end-to-end network policies called Propane. Propane uses regular expressions to define path constraints on allowed paths together with a preference operator that lets users describe the preferred paths. The inclusion of a preference operator allows Propane policies to refer to both intra-domain and inter-domain policies uniformly in the same language. We demonstrate how Propane makes it possible to define many common types of routing policies, such as those used in data centers and backbone networks. Given a high-level specification in the form of a Propane policy, we then show how to synthesize a collection of configurations for the distributed BGP routing protocol. The synthesized configurations are guaranteed to correctly implement the policy even when arbitrary combinations of link failures might occur. This result is surprising in a way, since there is no coordination between BGP routers beyond their routing advertisements. Key to making this possible are new data structures and static analysis algorithms for representing and analyzing the combined impact of the network policy and topology.

As with verification, utilizing abstraction to factor out network symmetries can greatly improve the scalability of configuration synthesis. We define a similar notion of network abstraction for the purposes of synthesis and demonstrate that making use of abstraction can improve performance by orders of magnitude while producing provably equivalent configurations. Abstraction, when used for synthesis, also has additional benefits such as guaranteed incremental configuration deployment. Namely, by having the user write additional abstraction invariants on the number of edges and nodes that may map to different abstract roles, it is possible to ensure the correctness of the generated configurations will continue to hold as the network evolves in any of a number of predefined ways.

7.1 Future Work and Open Problems

The work in this dissertation made progress towards the challenge of writing correct routing policy. However many challenges in this area yet remain.

7.1.1 Scalability

While many of the ideas in this dissertation have made control plane analysis significantly more scalable, dealing with large networks remains challenging in general. For example, fully verifying the networks from Section 4.8 remains out of scope for Minesweeper even when using abstraction. Part of the problem is the encoding for iBGP. Modeling a single destination at a time in the way Minesweeper does incurs a large cost when iBGP is used since the entire encoding must be duplicated. It may be possible to use state set representations such as BDDs [23] to represent all destinations simultaneously and avoid such overhead.

Yet another approach to scale control plane analysis may be to compute over- or under-approximations of the network behaviors rather than trying to always be precise. For example, abstract interpretation [34], the study of sound program analyses could possibly be used to compute an over-approximation of the number of network behaviors by trading off precision for efficiency.

Similarly, model-finding tools such as Alloy [98] can be used to under-approximate the network behaviors, potentially finding some (but not all) bugs. Similarly, bug finding techniques such as fuzzing [19, 52], which have been very successful in software bug finding, could potentially be used to quickly find many bugs in a network, even if they are not exhaustive.

7.1.2 Modularity

In the development of software, one rarely writes an entire program as a single function, or even a module/class. Instead, programs are decomposed into separate modules with separate concerns, and the modules present an interface to the outside world (*e.g.*, a collection of functions and their types) that hides many implementation details. Yet such modularity is less common in networks. In Propane, for example, it is assumed that modularity occurs only at the AS boundary. There may be future work in decomposing a network into modular subcomponents and then verifying/synthesizing these subcomponents separately in a way that preserves end-to-end behavior. For instance, checking internal reachability in a data center may depend on the routes advertised from a backbone network, which in turn depends on the data center. It may be possible to obtains guidance from rely-guarantee-style reasoning [65] in the software world, which is used to prove properties in a modular fashion when such dependencies are involved.

7.1.3 Quantitative Properties

To date, almost all work on verification and synthesis of the control plane, including this thesis, has focused on forwarding-based properties (*e.g.*, reachability or loops). Relatively little work has explored the possibility of reasoning about quantitative properties of the network such as load distribution, latency, bandwidth etc. Yet many network outages express themselves in quantitative ways (*e.g.*, a link becomes overloaded). Such reasoning would likely require a very different network model. For instance it may be possible to formalize a model of the network control plane in terms of probability distributions over stable trees rather than stable trees themselves. While

there has been some initial work in modeling the the data plane probabilistically [46], applying such an approach to the control plane would likely pose many challenges.

7.1.4 New Control Plane Languages

Many of the challenges for verification and synthesis of the network control plane arise due to the limitations and complexities of existing routing protocols. Software Defined Networking offers a programmatic interface to the data plane, but requires centralized control over the entire network. Given the recent emergence of programmable switch hardware [22], it may be appealing to be able to derive new routing protocols based on the needs of the network. For instance, Propane must limit the kinds of policies that are implementable based on limitations of BGP (*e.g.*, can only export a single best route). Further, new protocols may have different use cases, such as Hula [66], which optimizes dynamically for path utilization. Having a unified language and framework for defining, verifying, and synthesizing routing protocols and their policies may prove to be very useful moving forward.

Appendix A

Appendix

A.1 Proof of CP-equivalence

Here we give the full proof of CP-equivalence from Section 4.4. The proof requires additional lemmas and definitions not introduced in Section 4.4. First, we make an observation about attribute equality ($\approx$).

Theorem A.1.1. *Given an effective abstraction,* $\forall a, b.\ a \approx b \iff h(a) \approx h(b)$

Proof: *Immediate from rank-equivalence. Suppose* $a \approx b$*. Then* $a \not\prec b \wedge b \not\prec a$*. From rank-equivalence, this means that* $h(a) \not\prec h(b) \wedge h(b) \not\prec h(a)$*, and thus* $h(a) \approx h(b)$*. The reverse holds by the same reasoning.* $\square$

Theorem A.1.2. *Given an effective abstraction,* $\forall a.\ a = \bot \iff h(a) = \bot$

Proof: *We show one direction, but the argument is symmetric. Assume we know that* $h(a) = \bot$ *and* $a \neq \bot$*. From drop-ordering, we know that* $a \prec \bot$*, and from rank-equivalence we therefore know that* $h(a) \prec \bot$*. But this means that* $h(a) \neq \bot$ *(otherwise* $\prec$ *would not be a partial order).* $\square$

Next, we define *choice-equivalence*, which states that nodes in the abstract and concrete networks receive similar types of choices from similar neighbors:

Definition A.1.1. *We say that an abstraction* (f, h) *is* choice-equivalent *if the following holds:*

1. $\forall e, a.\ (e, a) \in \mathsf{choices}_{\mathcal{L}}(u) \implies (f(e), h(a)) \in \widehat{\mathsf{choices}}_{\mathcal{L}}(f(u))$
2. $\forall E, A.\ (E, A) \in \widehat{\mathsf{choices}}_{\mathcal{L}}(f(u)) \implies \forall e \mapsto E,\ \exists a.\ a \mapsto A \ \wedge\ (e, a) \in \mathsf{choices}_{\mathcal{L}}(u)$

Theorem A.1.3. *If we have a self-loop-free* SRP *and* $\widehat{SRP}$*, and an effective abstraction that is choice-equivalent, then the abstraction is label-equivalent.*

Proof: *Looking at the definition of* $\mathcal{L}$*, there are 3 cases to consider. First we observe that if* $v = d$*, then* $\mathcal{L}(d) = a_d$*. It follows that* $\widehat{\mathcal{L}}(f(d)) = \widehat{\mathcal{L}}(\widehat{d}) = \widehat{a}_d = h(a_d) = h(\mathcal{L}(d))$*. In the second case, using choice-equivalence and* $\forall\exists-$abstraction*, we can see that* $\mathsf{attrs}_{\mathcal{L}}(v) = \emptyset \iff \widehat{\mathsf{attrs}_{\mathcal{L}}}(f(v)) = \emptyset$*. Thus,* $h(\mathcal{L}(v)) = h(\bot) = \bot = \widehat{\mathcal{L}}(f(v))$*. For the final case with* $\mathsf{attrs}_{\mathcal{L}}(v) \neq \emptyset$*, we show the implications separately.*

Case (*$\Rightarrow$*) *Assume* $\mathcal{L}(v) = a$*. By the definition of* $\mathcal{L}$*, we know that* $a \in \mathsf{attrs}_{\mathcal{L}}(v)$ *and is minimal by* $\prec$*. We know that there is some edge* e *such that* $(e, a) \in \mathsf{choices}_{\mathcal{L}}(v)$*. Consider all concrete edges* $(e', a') \in \mathsf{choices}_{\mathcal{L}}(v)$*. From choice-equivalence, we know that* $(f(e'), h(a')) \in \widehat{\mathsf{choices}}_{\mathcal{L}}(f(v))$ *for each such pair. From rank-equivalence, we know* $(f(e), h(a)) \in \widehat{\mathsf{choices}}_{\mathcal{L}}(f(v))$ *and is minimal by* $\widehat{\prec}$*. By the definition of* $\mathcal{L}$*, we then know that* $\widehat{\mathcal{L}}(f(v)) = A = h(a)$*. By transitivity,* $\widehat{\mathcal{L}}(f(v)) = h(\mathcal{L}(v))$

Case (*$\Leftarrow$*) *Assume* $\widehat{\mathcal{L}}(f(v)) = A$*. We know that* $A \in \widehat{\mathsf{attrs}_{\mathcal{L}}}(f(v))$ *and is minimal by* $\widehat{\prec}$*. Assume* $(E, A) \in \widehat{choices}(f(v))$*. Consider all such* $(E', A') \in \widehat{\mathsf{choices}}_{\mathcal{L}}(f(v))$*. From choice-equivalence, we know that for any concrete edge* $e \mapsto E$*, there exists* a *such that* $h(a) = A$ *and* $(e, a) \in choices(v)$*. Therefore,* $a \in \mathsf{attrs}_{\mathcal{L}}(v)$*. Let us consider a smallest such* a *in terms of* $\prec$*. From rank-equivalence, we know that* a *is smaller than any* (e', a') *where* $f(e') \neq f(e)$ *since* A *was the smallest such value in* $\widehat{choices}$*. Therefore,* $a \in attrs(v)$ *and is minimal by* $\prec$*. Finally, we obtain that the labeling can be* a *(*$\mathcal{L}(v) = a$*). It follows from transitivity that* $h(\mathcal{L}(v)) = \widehat{\mathcal{L}}(f(v))$*.* $\square$

Theorem A.1.4. *If we have a self-loop-free SRP and $\widehat{SRP}$ and an effective abstraction that is choice-equivalent, then the abstraction is fwd-equivalent.*

Proof: *From Theorem A.1.3, we know that we have label-equivalence.*

Case 1 *We assume $e = (u, v) \in \mathsf{fwd}_{\mathcal{L}}(u)$ and need to show that $f(e) \in \widehat{\mathsf{fwd}}_{\widehat{\mathcal{L}}}(f(u))$. By the definition of $\mathsf{fwd}_{\mathcal{L}}$, we know that $\exists a.(e, a) \in \mathsf{choices}_{\mathcal{L}}(u) \wedge a \approx \mathcal{L}(u)$. From choice-equivalence, this means that*

$$(f(e), h(a)) \in \widehat{\mathsf{choices}}_{\mathcal{L}}(f(u))$$

Thus, because we have choice-equivalence, we have choice-equivalence. Recall from label equivalence: $h(\mathcal{L}(u)) = \widehat{\mathcal{L}}(f(u))$. From Theorem A.1.1, we have $a \approx \mathcal{L}(u)$ so $h(a) \approx h(\mathcal{L}(u))$ and thus $h(a) \approx \widehat{\mathcal{L}}(f(u))$. Then, by the definition of $\widehat{\mathsf{fwd}}_{\widehat{\mathcal{L}}}$:

$$f(e) \in \widehat{\mathsf{fwd}}_{\mathcal{L}}(f(u))$$

Case 2 *We will assume $(\widehat{u}, \widehat{v}) \in \widehat{\mathsf{fwd}}_{\mathcal{L}}(\widehat{u})$ and show that for all concrete nodes $u \mapsto \widehat{u}$, there exists a $v \mapsto \widehat{v}$ such that $(u, v) \in \mathsf{fwd}_{\mathcal{L}}(u)$. By the definition of $\mathsf{fwd}_{\mathcal{L}}$, we know that: $\exists A.\ ((\widehat{u}, \widehat{v}), A) \in \widehat{\mathsf{choices}}_{\mathcal{L}}(\widehat{u}) \wedge \widehat{\mathcal{L}}(\widehat{u}) \approx A$. From choice-equivalence, this means:*

$$\forall e \mapsto E,\ \exists a.\ h(a) = A \wedge (e, a) \in \mathsf{choices}_{\mathcal{L}}(u) \wedge \widehat{\mathcal{L}}(\widehat{u}) \approx A$$

Consider any such $e = (u, v)$ where $f(e) = E$. Rewriting slightly, we get:

$$\exists a.\ (e, a) \in \mathsf{choices}_{\mathcal{L}}(u) \wedge \widehat{\mathcal{L}}(f(u)) \approx h(a)$$

Once again, from Theorem A.1.1 and transfer-equivalence, we know that $\widehat{\mathcal{L}}(f(u)) = h(\mathcal{L}(u))$ *and so* $h(a) \approx h(\mathcal{L}(u))$*, and therefore:* $a \approx \mathcal{L}(u)$*. Finally, from the definition of* $\mathsf{fwd}_{\mathcal{L}}$ *we have*

$$e \in \mathsf{fwd}_{\mathcal{L}}(u)$$

.

□

Theorem A.1.5. *The forwarding behavior for any solution* $\mathcal{L}$ *to a well-formed, loop-free SRP will form a DAG rooted at the destination* d*.*

Proof: *We know that the solution is loop-free so the result must not have cycles. Also, there can only be one root for the DAG (*d*) because if there were another* d'*, then* $\mathcal{L}(d') = \bot$*, otherwise* d' *would forward to some neighbor. However, because the SRP is non-spontaneous, this can not happen.* □

Theorem A.1.6. *A well-formed, loop-free* SRP *and its effective abstraction* $\widehat{SRP}$ *are label- and fwd-equivalent. That is, for any* $\mathcal{L}$ *there exists label and fwd-equivalent* $\widehat{\mathcal{L}}$ *and vice-versa.*

Proof: *It suffices to first show choice-equivalence. We then get label-equivalence for free from Theorem A.1.3, and then that* SRP *and* $\widehat{SRP}$ *are fwd-equivalent from Theorem A.1.4.*

Because we know the SRP is loop-free and non-spontaneous, we know that any stable solution $\mathcal{L}$ *(and* $\widehat{\mathcal{L}}$*) must form a rooted DAG at the destination* d *(Theorem A.1.5). We start by showing a slightly strengthened inductive hypothesis: with the choice-equivalence property above for the subgraph corresponding to the actual forwarding edges in the provided concrete (or abstract) solution* $\mathcal{L}$ *(or* $\widehat{\mathcal{L}}$*). That is, given a concrete solution* $\mathcal{L}$ *we will only consider edges* $e = (u, v)$ *where* e *goes from level* $k + 1$ *to level* k *in the DAG, and similarly for the abstract network, we will only consider the corresponding edges* $f(e)$*. Symmetrically, for the other direction, we will only consider abstract edges* $\hat{e} = (\widehat{u}, \widehat{v})$ *going from level* $k + 1$ *to level* k *and only the edges* e *where* $f(e) = \widehat{e}$ *for the concrete network. For both directions, we will show label-equivalence*

($h(\mathcal{L}(v)) = \widehat{\mathcal{L}}(f(v))$) holds at each node. We show each direction of the stronger implication separately, using induction on the level of the DAG.

Base case (for $\Rightarrow$ and $\Leftarrow$): *For the base case, from the definition of $\mathcal{L}$, we know that $\mathcal{L}(d) = a_d$ and $\widehat{\mathcal{L}}(\widehat{d}) = \widehat{a}_d$. From dest-equivalence, we know that $f(d) = \widehat{d}$, so:*

$$\widehat{\mathcal{L}}(f(d)) = \widehat{\mathcal{L}}(\widehat{d}) = \widehat{a}_d = h(a_d) = h(\mathcal{L}(d))$$

Since there are no edges e going to a lower level in the DAG (than the root) in either the concrete or abstract, we are done.

Inductive case ($\Rightarrow$) *We are given $\mathcal{L}$ and show that there exists a $\widehat{\mathcal{L}}$ for the subgraph we have induced that has label-equivalence. Consider an arbitrary node u at depth k. Now, suppose $(e, a) \in \mathsf{choices}_{\mathcal{L}}(u)$ and $e = (u, v)$. We know that v appears at level $k - 1$ in the DAG. We also know that $a = \mathsf{trans}(e, \mathcal{L}(v)) \neq \bot$. Since $a \neq \bot$, from Theorem A.1.2, we know that $h(a) \neq bot$. By the IH with label-equivalence, we know that $\widehat{\mathcal{L}}(f(v)) = h(\mathcal{L}(v))$. From transfer-equivalence, we know that*

$$\widehat{\mathsf{trans}}(f(e), h(\mathcal{L}(v))) = h(\mathsf{trans}(e, \mathcal{L}(v))) = h(a) \neq \bot$$

By transitivity and label-equivalence (IH) then, we know:

$$\widehat{\mathsf{trans}}(f(e), \widehat{\mathcal{L}}(f(v))) = h(a)$$

By the definition of $\mathsf{choices}_{\mathcal{L}}$, it follows that

$$(f(e), h(a)) \in \widehat{\mathsf{choices}_{\mathcal{L}}}(f(u))$$

Hence, we have choice-equivalence. This means that the set of choices available at $f(u)$ from $f(v)$ is "the same" as the set of choices available at u from v. Since we have choice-equivalence, it follows that we have label-equivalence and fwd-equivalence for the subgraph under consideration.

Inductive case (⇐) *We are given $\widehat{\mathcal{L}}$ and show that there exists a $\mathcal{L}$ for the induced subgraph that has label-equivalence. Consider an arbitrary node $\widehat{u}$ at depth k of the abstract subgraph. Now, suppose $(E, A) \in \widehat{\mathsf{choices}}_{\mathcal{L}}(\widehat{u})$ and $E = (\widehat{u}, \widehat{v})$. From the $\forall\exists-$*abstraction *and the fact that f is onto, we know there must be at least some e such that $f(e) = E = (f(u), f(v))$ (otherwise E could not have been an abstract edge). Consider an arbitrary such $e = (u, v)$. We know that $\widehat{v}$ appears at level $k - 1$ in the DAG (and so does v by construction). We also know that $A = \widehat{\mathsf{trans}}(f(e), \widehat{\mathcal{L}}(f(v)) \neq \bot$. From Theorem A.1.2, we know that any a where $h(a) = A$ and $A \neq \bot$ has $a \neq \bot$. As before, we observe by the IH that $\widehat{\mathcal{L}}(f(v)) = h(\mathcal{L}(v))$. And so:*

$$A = \widehat{\mathsf{trans}}(f(e), h(\mathcal{L}(v))) = h(\mathsf{trans}(e, \mathcal{L}(v))) \neq \bot$$

Let a stand for $\mathsf{trans}(e, \mathcal{L}(v))$. Then $a = \mathsf{trans}(e, \mathcal{L}(v))$ and $h(a) = A$. By the definition of $\mathsf{choices}_{\mathcal{L}}$, it follows that:

$$\exists a.\ h(a) = A \wedge (e, a) \in \mathsf{choices}_{\mathcal{L}}(u)$$

Because we showed choice equivalence for any such edge e from any node $u \mapsto \widehat{u}$, we have choice-equivalence. This implies that we also have label- and fwd-equivalence.

Other edges *All that remains is to show that edges going to a equal or higher level of the DAG do not change the existing solution. Suppose we were given the concrete network. Consider such an edge $\widehat{e} = (\widehat{u}, \widehat{v})$. For this edge to affect the current solution $\widehat{\mathcal{L}}$, it must be the case that for some $\widehat{e}'$ and $\widehat{v}'$:*

$$\widehat{\mathsf{trans}}(\widehat{e}, \widehat{\mathcal{L}}(\widehat{v})) \mathrel{\widehat{\prec}} \widehat{\mathcal{L}}(\widehat{u}) = \widehat{\mathsf{trans}}(\widehat{e}', \widehat{\mathcal{L}}(\widehat{v}'))$$

Rewriting slightly:

$$\widehat{\mathsf{trans}}(\widehat{e}, h(\mathcal{L}(v))) \mathrel{\widehat{\prec}} \widehat{\mathsf{trans}}(\widehat{e}', h(\mathcal{L}(v'))))$$

From transfer equivalence:

$$h(\mathsf{trans}(e, \mathcal{L}(v))) \prec h(\mathsf{trans}(e', \mathcal{L}(v')))$$

From rank-equivalence:

$$\mathsf{trans}(e, \mathcal{L}(v)) \prec \mathsf{trans}(e', \mathcal{L}(v'))$$

Given the definition of $\mathcal{L}$, this leads to a contradiction with the fact that the concrete solution was indeed stable. In particular, node u has a better option through v' over v, and hence, the labelling is incorrect. We can conclude then, that here can be no such better option in the abstract network. The argument is symmetric for the other direction. □

Using Theorem A.1.6, we may conclude that any effective abstractions of common protocols, which produce loop-free routing, are CP-equivalent. Now we show that static routing, which is not necessarily loop-free, also has this property.

Theorem A.1.7. *Given self-loop-free SRP and $\widehat{SRP}$ for static routing with an effective abstraction, then it is fwd-equivalent.*

Proof: *Because the labeling at each node does not depend on the labeling at other nodes, the proof is direct. As before, we show choice-equivalence, then rely on A.1.4 to derive CP-equivalence.*

Case (⇒) *Assume $e = (u, v)$. We have $(e, a) \in \mathsf{choices}_{\mathcal{L}}(u)$. By unfolding the definition of $\mathsf{choices}_{\mathcal{L}}$, we know that $a = \mathsf{trans}(v, \mathcal{L}(v))$. By transfer equivalence, we know that*

$$h(a) = h(\mathsf{trans}(e, \mathcal{L}(v))) = \widehat{\mathsf{trans}}(f(e), h(\mathcal{L}(v)))$$

There are now 2 cases. Suppose $a = 1$. Then $h(a) = 1$, so

$$\widehat{\mathsf{trans}}(f(e), h(\mathcal{L}(v))) = 1$$

The definition of trans *does not depend on the attribute for static routes, we know that :*

$$\widehat{\mathsf{trans}}(f(e), \widehat{\mathcal{L}}(v)) = 1$$

It follows that $(f(e), 1) \in \mathsf{choices}_{\mathcal{L}}(f(u))$

The case for $a = 0$*, is symmetric.*

Case (⇐) *Suppose* $((\widehat{u}, \widehat{v}), A) \in \mathsf{choices}_{\mathcal{L}}(\widehat{u})$*. We need to show that for any edge* $e \mapsto E$*, there exists an* a *such that* $(e, a) \in \mathsf{choices}_{\mathcal{L}}(u)$ *and* $h(a) = A$*. Let us choose* $a = A$ *for static routes. Clearly* $h(a) = A$ *since* h *is the identity. Consider arbitrary edge* $e \mapsto E$*. We have:*

$$\widehat{\mathsf{trans}}(f(e), \widehat{\mathcal{L}}(f(v)))) = A = h(a) = a$$

Again, since the definition of $\widehat{\mathsf{trans}}$ *does not depend on the neighbor attribute, we can replace it with any value. In particular, this is the same as:*

$$a = \widehat{\mathsf{trans}}(f(e), h(\mathcal{L}(v)))$$

From transfer-equivalence and transitivity we know that:

$$a = \mathsf{trans}(e, \mathcal{L}(v))$$

Finally, from the definition of $\mathsf{choices}_{\mathcal{L}}$*:* $\exists a.\ h(a) = A \wedge (e, a) \in \mathsf{choices}_{\mathcal{L}}(u)$ □

Corollary A.1.8. *Suppose we have a self-loop-free* SRP *and* $\widehat{SRP}$ *for RIP, OSPF, static routing, or BGP (without loop prevention), related by effective abstraction* (f, h)*. There is a solution* $\mathcal{L}$*, where each node* $u_1 \mapsto \widehat{u_1}$ *forwards along label path* $s = \mathcal{L}(u_1) \ldots \mathcal{L}(u_k)$ *to some node* $u_k \mapsto \widehat{u_k}$ *iff there is a solution* $\widehat{\mathcal{L}}$ *that forwards along the label path* $\widehat{s} = \mathcal{L}(\widehat{u_1}) \ldots \mathcal{L}(\widehat{u_k})$ *and* $h(s) = \widehat{s}$*.*

Proof: *We show each direction separately.*

Case (⇒) *Suppose $\mathcal{L}$ is a solution for SRP. Given any two nodes u and v where u can reach v, there exists a path $p = u, w_1, \ldots, w_k, v$ where $(u, w_1) \in \mathsf{fwd}_{\mathcal{L}}(u)$ and $(w_i, w_{i+1}) \in \mathsf{fwd}_{\mathcal{L}}(w_i)$ and $(w_k, v) \in \mathsf{fwd}_{\mathcal{L}}(w_k)$. Because $\mathcal{L}$ and $\widehat{\mathcal{L}}$ are fwd-equivalent, we know that $(f(u), f(w_1)) \in \widehat{\mathsf{fwd}}_{\widehat{\mathcal{L}}}(f(u))$ and so on. Therefore, there is an abstract path in $\widehat{\mathcal{L}}$ where $f(u)$ can reach $f(v)$ where the path has the form $f(u), f(w_1), \ldots, f(w_k), f(v)$. The labels of the concrete path are $s = \mathcal{L}(u), \mathcal{L}(w_1), \ldots, \mathcal{L}(w_k), \mathcal{L}(v)$. Similarly, the abstract path has labels $\widehat{\mathcal{L}}(f(u)), \ldots, \widehat{\mathcal{L}}(f(v))$. It follows from label-equivalence that $\widehat{\mathcal{L}}(f(u)), \ldots, \widehat{\mathcal{L}}(f(v)) = h(\mathcal{L}(u)), \ldots, h(\mathcal{L}(v))$. Finally, the definition of h gives us: $h(\mathcal{L}(u)), \ldots, h(\mathcal{L}(v)) = h(s)$*

Case (⇐) *Symmetric to the first case. Suppose $\widehat{L}$ is a solution for $\widehat{SRP}$. Consider an arbitrary path $\widehat{u}, \widehat{w_1}, \ldots, \widehat{w_k}, \widehat{v}$. Then we know $(\widehat{u}, \widehat{w_1}) \in \widehat{\mathsf{fwd}}_{\mathcal{L}}(\widehat{u})$ and so on. From the fact that $\mathcal{L}$ and $\widehat{\mathcal{L}}$ are fwd-equivalent, every node u where $u \mapsto \widehat{u}$ will follow some path $(u, w_1) \in fwd(u)$ and so on. Therefore, there will be a concrete path $u, w_1, \ldots, w_k, v$ such that $w_i \mapsto \widehat{w_i}$, and $v \mapsto \widehat{v}$. The abstract path $\widehat{s} = \widehat{\mathcal{L}}(\widehat{u}), \ldots, \widehat{\mathcal{L}}(\widehat{v})$. Similarly, the concrete path will have $s = \mathcal{L}(u), \ldots, \mathcal{L}(v)$. To show that $\widehat{s} = h(s)$, we simply use label-equivalence: $h(s) = h(\mathcal{L}(u)), \ldots, h(\mathcal{L}(v)) = \widehat{s}$.*

□

Theorem A.1.9. *If a well-formed SRP and $\widehat{SRP}$ for BGP has an $\forall\forall-$*abstraction *and is transfer-approx, then for all solutions $\mathcal{L}$ to SRP, and for all abstract nodes $\widehat{u} \in \widehat{V}$, $|\mathcal{B}_{\mathcal{L}}(\widehat{u})| \leq |\mathsf{prefs}(\widehat{u})|$.*

Proof: *Because we have rank-equivalence and an $\forall\forall-$*abstraction*, the only way two nodes will forward to different neighbors is the transfer functions are different. Otherwise, both nodes would receive the same $\mathsf{choices}_{\mathcal{L}}$ as in Theorem A.1.6 and because of the universal abstraction, they both have an edge to the best such choice and will use this neighbor. Due to relative-transfer-equivalence, the only time this can occur is when two nodes have different transfer functions due to loop prevention.*

First we show that there can be $|\mathsf{prefs}(\widehat{u})|$ different behaviors. Consider the example in Figure 4.8. In the example, $\widehat{u}$ has a local preference for $\widehat{v_1}$ over $\widehat{v_2}$ over $\widehat{v_3}$ etc. In this case, $|\mathsf{prefs}(\widehat{u})| = 3$. There is a stable solution where u_1 forwards to v_{11} since that is the best path.

u_2 would prefer to use this path, but cannot because it is already on the path, so it cannot consider v_{11} due to its transfer function. Instead, u_2 will use the next best choice v_{21}. Similarly, u_3 would like to use v_{11} or v_{21} but cannot due to loops. Therefore, u_3 will forward to v_{31} instead.

Because there is a universal abstraction (full mesh), and because we have rank-equivalence and relative-transfer-equivalence, each node has the same choices modulo loops. Such a chain as shown in Figure 4.8 is the only way we can get such different behavior. Now we show that there can not be more than $|\mathsf{prefs}(\widehat{u})|$ behaviors. The proof is by contradiction. Suppose we have another node u_4 and u_4 will forward to a different node that each of u_1 through u_3. u_4 can not continue the chain by falling back to the next lowest local preference since all local preferences have been exhausted by u_1 through u_3. Therefore, u_4 will forward to one of the same neighbors as u_1 through u_3. But this contradicts the assumption. Therefore, there can not be more than $|\mathsf{prefs}(\widehat{u})|$ behaviors.

□

Theorem A.1.10. *Suppose we have well-formed SRP, $\widehat{SRP}$, and $\overline{SRP}$ for BGP with an effective abstraction (f, h). For any solution $\mathcal{L}$ to SRP, there exists a refinement $(f_r, h_r) \sqsubseteq_{(f_s,h_s)} (f, h)$ where $\overline{\mathcal{L}}$ is a solution to $\overline{SRP}$, and $\mathcal{L}$ and $\overline{\mathcal{L}}$ are label- and fwd-equivalent.*

Proof: *First, we will show a particular refinement. From Theorem A.1.9, we know that any solution to $\mathcal{L}$ can only have $|\mathsf{prefs}(\widehat{v})|$ behaviors. Let use define $f_r(v) = \overline{v} = \mathsf{behavior}(f(v))(i)$, where this notation means that we pick out the ith node in $\overline{V}$ such that f_s maps it to $\widehat{v}$. We can modify this scheme slightly to ensure that f_r is an* onto *function, if no node would map to the kth behavior, then pick an arbitrary node that maps to the jth behavior (if there is more than one node that maps to the jth behavior), and map it to the kth behavior instead. This is a valid refinement to (f, h) since $f = f_s \circ f_r$ and f_r is onto.*

Since we have a particular solution $\mathcal{L}$ that is loop-free (since BGP is loop-free), we know all the edges in SRP that are not used due to loops. For example, in Figure 4.9 in the concrete network (left), the green node would have transfer function $\perp$ from each of the red neighbors below due to loop prevention.

Consider an isomorphic network G', where all such edges are removed (e.g., directed edges from the green to red nodes). Similarly, in the refined network $\overline{G}'$, we would remove the corresponding edges (e.g., the directed edge from the green to red nodes).

*The particular refinement f_r we chose is important because we will still have an $\forall\forall-$*abstraction *after removing these edges since each node with such unique behavior rejected the same nodes (due to loops) to accept the worse path. Therefore, removing the abstract edge and concrete edges remains a universal abstraction.*

*The same solution $\mathcal{L}$ is a solution for the isomorphic SRP where the loop-prevention mechanism for BGP is removed (i.e., we don't block paths with loops). Since we have relative-transfer-equivalence, by removing the loop condition, we get full transfer-equivalence. We can then simply appeal to Theorem A.1.6 to derive $\exists\forall-$*equivalent *and preference-equivalence of $\mathcal{L}$ and $\overline{\mathcal{L}}$ for the isomorphic networks.*

Finally, if we add back the abstract edges that we removed, we need to show that we still have the same solution $\overline{\mathcal{L}}$ with loop-prevention. We do this by showing that such edges would be rejected as loops. Given that we have CP-equivalence, and in the concrete solution $\mathcal{L}$ this edge would result in a loop of the form $u, w_1, \ldots, w_k, u$, we know that the abstract path would also have a loop $f_r(u), f_r(w_1), \ldots, f_r(w_k), f_r(u)$. □

Next we show the other direction. Note that in both cases, the proof is constructive and thus responsible for identifying the particular appropriate refinement (f_r, h_r) and (f_s, h_s).

Theorem A.1.11. *Suppose we have well-formed SRP, $\widehat{SRP}$, and $\overline{SRP}$ for BGP with an effective abstraction (f, h). For any solution $\overline{\mathcal{L}}$ to $\overline{SRP}$, then there exists a refinement $(f_r, h_r) \sqsubseteq_{(f_s,h_s)} (f, h)$ where $\mathcal{L}$ is a solution to SRP, and $\mathcal{L}$ and $\overline{\mathcal{L}}$ are label- and fwd-equivalent.*

Proof: *Setup f_r such that, for each node $\overline{v}$ that forwards for an attribute that is not the best when ignoring loop-prevention, we have a single node in $v \in V$ map to such a $\overline{v}$. For every other node $\overline{v}$, that forwards to the best available option, we map every other v to each of these $\overline{v}$. That is,*

we assign a single concrete node for each unique behavior that is not the best route and all other nodes map to the abstract node that has the best route.

*As before, we remove each edge in $\overline{SRP}$ that corresponds to an edge rejected due to loops in $\overline{\mathcal{L}}$, and all corresponding concrete edges related under f_r. As before, in the concrete network, we will still have a $\forall\forall-$*abstraction *since each concrete node that forwards to a non-best path does so because the better paths are rejected due to loops.*

This network will have the same solution $\overline{\mathcal{L}}$ but has full transfer-equivalence, and we can again appeal to Theorem A.1.6 for preference- and CP-equivalence for BGP without loop-prevention.

If we add back the abstract edges that we removed, we need to show that the same solution $\mathcal{L}$ is still a solution with loop-prevention. Suppose that the abstract node $\overline{u}$ prevented a loop $\overline{u}, \overline{w}_1, \ldots, \overline{w}_k, \overline{u}$. Then each node u where $f_r(u) = \overline{u}$ that chose a non-best path in the concrete network also did so due to loop-prevention. □

Corollary A.1.12. *Suppose we have well-formed SRP, $\widehat{SRP}$, and $\overline{SRP}$ for BGP with an effective abstraction (f, h). There is a solution $\mathcal{L}$, where each node $u_1 \mapsto \widehat{u_1}$ forwards along path $s = \mathcal{L}(u_1) \ldots \mathcal{L}(u_k)$ to some node $u_k \mapsto \widehat{u_k}$ iff there is a solution $\overline{\mathcal{L}}$ where each node $\overline{u_1} \mapsto \widehat{u_1}$ forwards along path $\overline{s} = \mathcal{L}(\overline{u_1}) \ldots \mathcal{L}(\overline{u_k})$ to some $\overline{u_k} \mapsto \widehat{u_k}$ such that $h(s) = h_s(\overline{s})$.*

Proof: *We show each direction separately.*

***Case (⇒)** Suppose $\mathcal{L}$ is a solution for SRP. From Theorem A.1.10, we know there exists a refinement $(f_r, h_r) \sqsubseteq (f, h)$ of for $\overline{SRP}$ with solution $\overline{\mathcal{L}}$, and also that $\mathcal{L}$ and $\overline{\mathcal{L}}$ are fwd-equivalent. Given any two nodes u and v where u can reach v, there exists a path $p = u, w_1, \ldots, w_k, v$ where $(u, w_1) \in \mathsf{fwd}_{\mathcal{L}}(u)$ and $(w_i, w_{i+1}) \in \mathsf{fwd}_{\mathcal{L}}(w_i)$ and $(w_k, v) \in \mathsf{fwd}_{\mathcal{L}}(w_k)$. Because $\mathcal{L}$ and $\overline{\mathcal{L}}$ are fwd-equivalent, we know that $(f_r(u), f_r(w_1)) \in \overline{\mathsf{fwd}_{\mathcal{L}}}(f_r(u))$ and so on. Therefore, there is an abstract path in $\overline{\mathcal{L}}$ where $f_r(u)$ can reach $f_r(v)$ of the form $f_r(u), f_r(w_1), \ldots, f_r(w_k), f_r(v)$. Since f_r is* onto *from Theorem A.1.10, we know that this is the case for every $f_r(u) \in f_s^{-1}(f(u))$. Observe that the labels of the concrete path are $s = \mathcal{L}(u), \mathcal{L}(w_1), \ldots, \mathcal{L}(w_k), \mathcal{L}(v)$. Similarly, the*

abstract path has labels $\overline{\mathcal{L}}(f_r(u)), \ldots, \overline{\mathcal{L}}(f_r(v))$. *Due to label-equivalence, this is the same as* $h_r(\mathcal{L}(u)), \ldots, h_r(\mathcal{L}(v))$, *which is just* $h_r(s)$.

Recall that we must show that $h_s(\overline{s}) = h(s)$ *Since we know* $h_r(s) = \overline{s}$, *we have* $h_s(h_r(s)) = h(s)$. *Finally, because* $h = h_s \circ h_r$, *these are equivalent.*

Case (⇐) *Suppose* $\overline{L}$ *is a solution for* $\overline{SRP}$, *then from Theorem A.1.10, we know that there exists an onto refinement* $(f_r, h_r) \sqsubseteq (f, h)$ *where* $\mathcal{L}$ *and* $\overline{\mathcal{L}}$ *are* $\exists\forall-$equivalent. *Consider an arbitrary path* $\overline{u}, \overline{w_1}, \ldots, \overline{w_k}, \overline{v}$. *Then we know* $(\overline{u}, \overline{w_1}) \in \overline{\mathsf{fwd}_{\mathcal{L}}}(\overline{u})$ *and so on. From the fact that* $\mathcal{L}$ *and* $\overline{\mathcal{L}}$ *are* $\exists\forall-$equivalent, *every node* u *that maps to* $\overline{u}$ *forwards to the same neighbor. That is, we know that each* u *where* $f_r(u) = \overline{u}$, *has the same* w_1 *where* $f_r(w_1) = \overline{w_1}$ *and* $(u, w_1) \in \mathsf{fwd}_{\mathcal{L}}(u)$ *and so on. Therefore, each node* $u \in f_r^{-1}(\overline{u})$ *has the same path starting after* w_1*:* $u, w_1, \ldots, w_k, v$. *The abstract path has labels* $\overline{\mathcal{L}}(f_r(u)), \ldots, \overline{\mathcal{L}}(f_r(v))$. *Due to label-equivalence, this is the same as* $h_r(\mathcal{L}(u)), \ldots, h_r(\mathcal{L}(v))$, *which is* $h_r(s)$.

Once again, we have $h_s(h_r(s)) = h(s)$, *which follows from the fact that* $h = h_s \circ h_r$. □

A.2 Proof of Concretization Correctness

The rest of the appendix demonstrates the correctness of the synthesis of templates and their concretization by showing that compilation and concretization commute.

A.2.1 Proof Sketch

The goal of the proof is to show that the concretization and compilation functions commute. One can concretize an abstract policy and then compile the concrete result, or compile an abstract policy and the concretize the abstract result and know that the configurations will be the same. More specifically, we are interested in establishing the following theorem:

Theorem A.2.1. *for all Propane/AT policies pol, contexts* Γ*, and topologies G and* G^A *related by the homomorphism* f*,*

$$\mathrm{con}(\mathrm{compile}(pol, G^A), \Gamma, f, G) = \mathrm{compile}(\mathrm{con}(pol, \Gamma, f), G)$$

The proof proceeds in the following steps.

Product Graphs (Step 1): First, we establish several facts about the relationship of the concrete and abstract product graphs given the definition for building the product graph. We start by assuming that both an abstract constraint and its concretized form are compiled to product graphs and preference functions:

$$\begin{aligned} \mathrm{compile}_{\mathrm{PG}}(t_1 \Rightarrow \mathrm{con}(r_1, \Gamma, f) \cap \mathrm{end}(L) >> \ldots, G) &= (t_1, PG, \mathrm{pref}) \\ \mathrm{compile}_{\mathrm{PG}}(t_2 \Rightarrow r_1 >> \ldots >> r_n, G^A) &= (t_2, PG^A, \mathrm{pref}^A) \end{aligned}$$

where L can be thought of as a set of destination locations (possibly Σ) for the prefix. Using these assumptions, we show that several relationships hold between PG and PG^A as well as between the inferred preference functions pref and pref^A. In particular, we show that there is a homomorphism f_{pg} from PG to PG^A, and then using this fact, we demonstrate that the inferred local preference functions pref and pref^A have the property that for all nodes m in PG, $\mathrm{pref}(m) = \mathrm{pref}^A(f_{pg}(m))$.

Substitution of Constraints (Step 2): The next step is to use the relationships between PG and PG^A established in step 1 to show that concretization and compilation commute for an individual Propane/AT constraint p:

$$\mathrm{con}(\mathrm{compile}(p, G^A), \Gamma, f, G) = \mathrm{compile}(\mathrm{con}(p, \Gamma, f), G)$$

Constraints are of the form: $t \Rightarrow r_1 >> \ldots >> r_n$. The proof proceeds by case analysis on the predicate t. The proof is long, but mainly involves repeatedly applying the definition of concretization

and compilation until all abstract sets of locations are written in terms of concrete sets of locations. The proofs from step 1 are then used to show that these sets are equivalent.

Substitution of Policies (Step 3): Finally, this theorem is lifted to Propane/AT policies $pol = p_1, \ldots, p_n$ using the proof from part 2 for individual constraints to obtain the final theorem:

$$\text{con}(\text{compile}(\mathit{pol}, G^A), \Gamma, f, G) = \text{compile}(\text{con}(\mathit{pol}, \Gamma, f), G)$$

The proof is obtained by unfolding of the definition of compilation and concretization and applying the previous theorem.

A.2.2 Substitution Proof

Product Graphs (Step 1)

In this section, we will establish several relationships between the product graph and preference functions for abstract and concretized Propane/AT policies. To do so, we first relate the languages denoted by regular expressions under concretization. Then we lift this observation to automata, and finally to product graphs. We use these connections to construct a homomorphism for the product graphs and to relate the inferred BGP local preferences for abstract and concrete devices.

Lemma A.2.2. *For any context Γ, homomorphism f, and regular expression r, path $p \in \mathcal{L}(con(r, \Gamma, f)) \iff f(p) \in \mathcal{L}(r)$.*

Proof: By induction on the structure of r

Case $\emptyset$:

$$\begin{aligned} p \in \mathcal{L}(con(\emptyset, \Gamma, f)) &\iff f(p) \in \mathcal{L}(\emptyset) \\ p \in \emptyset &\iff f(p) \in \emptyset \end{aligned}$$

Case l:

$$
\begin{aligned}
p \in \mathcal{L}(con(l, \Gamma, f)) &\iff f(p) \in \mathcal{L}(l) \\
p \in \mathcal{L}(\Sigma\ f^{-1}(l)) &\iff f(p) \in \mathcal{L}(l) \\
p \in \bigcup f^{-1}(l) &\iff f(p) \in \mathcal{L}(l) \\
p \in f^{-1}(l) &\iff f(p) \in \{l\} \\
p \in f^{-1}(l) &\iff f(p) = l
\end{aligned}
$$

by the homomorphism

Case $r_1 \cup r_2$:

$$
\begin{aligned}
p \in \mathcal{L}(con(r_1 \cup r_1, \Gamma, f)) &\iff f(p) \in \mathcal{L}(r_1 \cup r_2) \\
p \in \mathcal{L}(con(r_1, \Gamma, f) \cup con(r_1, \Gamma, f)) &\iff f(p) \in \mathcal{L}(r_1 \cup r_2) \\
p \in \mathcal{L}(con(r_1, \Gamma, f)) \cup \mathcal{L}(con(r_1, \Gamma, f)) &\iff f(p) \in \mathcal{L}(r_1 \cup r_2) \\
p \in \mathcal{L}(con(r_1, \Gamma, f)) \vee p \in \mathcal{L}(con(r_1, \Gamma, f)) &\iff f(p) \in \mathcal{L}(r_1) \vee f(p) \in \mathcal{L}(r_2)
\end{aligned}
$$

by cases and the IH

Case $r_1 \cap r_2$:

symmetric to $\cup$ case with $\wedge$ instead of $\vee$

Case $!r$:

$$
\begin{aligned}
p \in \mathcal{L}(con(!r, \Gamma, f)) &\iff f(p) \in \mathcal{L}(!r) \\
p \in \mathcal{L}(!con(r, \Gamma, f)) &\iff f(p) \in \mathcal{L}(!r) \\
p \in \Sigma^* - \mathcal{L}(con(r, \Gamma, f)) &\iff f(p) \in \Sigma^* - \mathcal{L}(r)
\end{aligned}
$$

by the IH

Case r^*:

$$
\begin{aligned}
p \in \mathcal{L}(con(r^*, \Gamma, f)) &\iff f(p) \in \mathcal{L}(r^*) \\
p \in \mathcal{L}(con(r, \Gamma, f)^*) &\iff f(p) \in \mathcal{L}(r^*) \\
p \in \textstyle\bigcup_{i \in \mathbb{N}} \mathcal{L}(con(r, \Gamma, f))^i &\iff f(p) \in \textstyle\bigcup_{i \in \mathbb{N}} \in \mathcal{L}(r)^i
\end{aligned}
$$

Follows from IH and definition of $\mathcal{L}(r)^i$

□

For the proof, we need to make sure that automata are constructed in a particular way to ensure a product graph homomorphism will exist. We assume an invariant that when regular expressions are translated to finite automata, there is no transition back to the initial state q_0. This can be done easily by introducing a second copy of q_0 that allows transitions. The reason is to distinguish if a state corresponds to a router "owning" a prefix. We also compile automata in a particular way as defined below

Definition A.2.1. *Given a regular expression r over abstract locations, homomorphism f, a nonempty set of locations L, and a finite state machine $M^A = (\Sigma, Q^A, q_0, \sigma^A, F^A)$, we construct a concrete state machine $M = (\Sigma, Q, q_0, \sigma, F)$ for the concretized policy $con(r, \Gamma, f) \cap \mathrm{end}(L)$ in the following way:*

1. $\sigma(q_0, l) = q' \iff \sigma^A(q, f(l)) = q'$ ***for each*** $l \in L$
2. $\sigma(q, l) = q' \iff \sigma^A(q, f(l)) = q'$ ***for each*** $l \in \Sigma - L$
3. $q \in F \iff q \in F^A$

Next, we show that the construction is correct. That is, the machine recognizes exactly the language $con(r, \Gamma, f) \cap \mathrm{end}(L)$.

Lemma A.2.3. *for any path p, M matches $p \iff p \in \mathcal{L}(con(r, \Gamma, f) \cap \mathrm{end}(L))$*

Proof:

Case *($\Rightarrow$)*

Assume M matches p. Then this means that M^A matches $f(p)$. If we have a trace of string of the form $p = x_1 \cdot \cdots \cdot x_n$, then $f(p) = f(x_1) \cdot \cdots \cdot f(x_n)$. For each transition $\sigma(q, l) = q'$ in the concrete automaton, we have transition $\sigma(q, f(l)) = q'$ in the abstract automaton, and they have the same final states, so this trace is accepting for M^A. Because M^A matches $f(p)$, this means that $f(p) \in \mathcal{L}(r)$. From Lemma 1, this means that $p \in \mathcal{L}(con(r, \Gamma, f))$. We also know that, since

there is only a transition from the initial state $\sigma(q_0, l)$ defined for $l \in L$ by construction, it means that $p \in \mathcal{L}(end(L))$. Since we know that $p \in \mathcal{L}(con(r_i, \Gamma, f))$ and $p \in \mathcal{L}(end(L))$, we know that $p \in \mathcal{L}(con(r, \Gamma, f) \cap end(l))$ and thus $p \in \mathcal{L}(con(r, \Gamma, f) \cap \mathrm{end}(L))$.

Case *($\Leftarrow$)*

Assume $p \in \mathcal{L}(con(r, \Gamma, f) \cap \mathrm{end}(L))$. This means that $p \in \mathcal{L}(con(r, \Gamma, f))$ and $p \in \mathcal{L}(end(L))$. Lemma 1, we know that $f(p) \in \mathcal{L}(r)$, which means that M^A matches $f(p)$. Consider any string of the form $p = l \cdot x_2 \cdot \cdots \cdot x_n$ such that $p \in \mathcal{L}(con(r, \Gamma, f) \cap end(L))$. Clearly $l \in L$ and there is a transition defined for $\sigma(q_0, l) = q_1$. This means that M^A matches $f(l) \cdot f(x_2) \cdot \cdots \cdot f(x_n)$. For each transition $\sigma^A(q, f(x_i)) = q'$, we have $\sigma(q, x_i) = q'$ in M. Thus the string $l \cdot x_2 \cdot \cdots \cdot x_n$ is accepted along the same path of automaton states. □

Lemma A.2.4. *M and M^A have the same set of states. That is, $Q = Q^A$.*

Proof: *This follows from the construction of M. States in M are copied over from states in M^A.* □

Note: For the remainder of this section, we will prove a collection of lemmas assuming the following:

$$\begin{aligned} \mathrm{compile}_{\mathrm{PG}}(t_1 \texttt{=>} \mathrm{con}(r_1, \Gamma, f) \cap \mathrm{end}(L) \texttt{>>} \ \ldots, G) &= (t_1, PG, \mathrm{pref}) \\ \mathrm{compile}_{\mathrm{PG}}(t_2 \texttt{=>} r_1 \texttt{>>} \ \ldots \texttt{>>} r_n, G^A) &= (t_2, PG^A, \mathrm{pref}^A) \end{aligned}$$

We will also assume that $PG = (G', start, rank)$ and $PG^A = (G'^A, start^A, rank^A)$ and that either $L = \Sigma$, or $L = \{l\}$.

Lemma A.2.5. *For any node $n = (l, q_1, \ldots, q_k)$ in PG there exists a node $f(n) = (f(l), q_1, \ldots, q_k)$ in PG^A.*

Proof: *Consider a path of length k in PG to node n. The proof proceeds by induction on the path length k, if n is connected to $start$, then $q_i = q_{0_i}$. Since there are no transitions back to the initial state, there must be a node $f(n) = (f(l), q_{0_1}, \ldots, q_{0_1})$ or else there is a contradiction for Lemma 1. For the inductive case, if n is the kth hop along the path, then it has an inbound edge from neighbor $k-1$. Call this neighbor m. Then (m, n) is an edge in PG, there is a node $f(m)$ with the same states. From the graph homomorphism, we know that $(f(m), f(n))$ must be an edge in PG^A and since $f(m)$ has the same states as m, by the construction of the automata, $f(n)$ will have the same states as n.* □

Lemma A.2.6. *The homomorphism f can be lifted to a new homomorphism f_{pg} over the product graphs in the following way:*

$$\begin{aligned} f_{pg}(start) &= start^A \\ f_{pg}((l, q_1, \ldots, q_n)) &= (f(l), q_1, \ldots, q_n) \end{aligned}$$

Now we prove that $f_{pg} : G' \to G'^A$ is a valid homomorphism

Proof: *From Lemma 5, $(f(l), q_1, \ldots, q_n)$ is a valid node in PG'^A. Assume there is an edge in PG from $x = (l, q_1, \ldots, q_n)$ to $y = (l', q'_1, \ldots, q'_n)$. We know that $f(x) = (f(l), q_1, \ldots, q_n)$ and $f(y) = (f(l'), q'_1, \ldots, q'_n)$. From the construction of PG, we know that $\sigma_i(q_k, l) = q'_k$. Also from the construction of the abstract automata, we also know that $\sigma_i(q_k, l) = q'_k \iff \sigma_i^A(q_k, f(l)) = q'_k$. Therefore, we can conclude that $\sigma_i^A(q_k, f(l)) = q'_k$. From the definition of PG^A, this means that there must be an edge in PG^A for $(f(x), f(y))$.* □

Lemma A.2.7. *If $f_{pg}(n) = N$, then the product graph preference of these nodes is equal:* $\mathrm{rank}(n) = \mathrm{rank}^A(N)$.

Proof: *Assume $n = (l, q_1, \ldots, q_k)$. From Lemma 6, we know $f_{pg}(n) = (f(l), q_1, \ldots, q_k)$. We also know that $q_i \in F \iff q_i \in F^A$. By the definition of the ranking function* rank*, then* $\mathrm{rank}(n) = \{i \mid q_i \in F\} = \{i \mid q_i \in F^A\} = \mathrm{rank}^A(N)$. □

Next, we show that whenever there is a transition in PG^A, there is a corresponding step in the concrete product graph PG. This will allows to show that, if the preference inference succeeds for the abstract product graph, then it will also succeed for the concrete product graph while inferring the same preferences.

Lemma A.2.8. *If, for PG and PG^A we have the following:*

1. (a, b) *is an edge in the concrete topology G*
2. $\text{topo}(m) = a$
3. $f_{pg}(m) = M$
4. (M, N) *is an edge in* PG^A
5. $\text{topo}(N) = f(b)$

then there exists a node n in PG where:

1. $\text{topo}(n) = b$
2. $f_{pg}(n) = N$
3. (m, n) *is an edge in the product graph* PG

Proof:

Since $\text{topo}(m) = a$, *suppose that* $m = (a, q_1, \ldots, q_k)$. *We know that* $f_{pg}(m) = M = (f(a), q_1, \ldots, q_k)$. *By assumption* $N = (f(b), s_1, \ldots, s_k)$. *Let us take the node* $n = (b, s_1, \ldots, s_k)$.

1. Clearly $\text{topo}(n) = b$

2. Clearly $f_{pg}(n) = (f(b), s_1, \ldots, s_k) = N$

3. Since (M, N) *is an edge in* PG^A, *we know that, for each automata transition function:* $\sigma^A(q_i, f(b)) = s_i$. *From the automata construction, this means that for each regular expression,* $\sigma(q_i, b) = s_i$ *if* $q_i \neq q_{0_i}$. *We know that* $q_i \neq q_{0_i}$ *due to the side condition for automaton construction that ensures no transitions to the initial state. It follows that* (m, n) *is an edge in the product graph since* (a, b) *is a valid topology edge by assumption, and* $\sigma(q_i, b) = s_i$.

□

Lemma A.2.9. *For any labelled transition $m \xrightarrow{l} m'$ in PG, there is a corresponding transition $f(m) \xrightarrow{f(l)} f(m')$ in PG^A.*

Proof: *if $m' = (l, q_1, \ldots, q_n)$, then $f(m') = (f(l), q_1, \ldots, q_n)$. Since there is the edge (m, m') in PG, then there is the edge $(f(m), f(m'))$ in PG^A by the homomorphism. Because $\text{topo}(f(m')) = f(l)$, there must be a transition $f(m) \xrightarrow{f(l)} f(m')$.* □

Lemma A.2.10. *In the concrete product graph PG, $m \leq m' \iff f_{pg}(m) \leq f_{pg}(m')$ in the abstract product graph PG^A.*

Proof: *We show that $\leq$ forms a simulation relation for the subgraph reachable from m and m' in PG iff $\leq$ forms a simulation relation for the subgraph reachable from $f(m)$ and $f(m')$ for PG^A. This involves showing that $m \geq_{rank} m' \iff f(m) \geq_{rank} f(m')$, and that for every transition $m \xrightarrow{l} n$ in PG there is a transition $m' \xrightarrow{l} n'$ iff for every transition $f(m) \xrightarrow{f(l)} f(n)$ in PG^A there is a transition $f(m') \xrightarrow{f(l)} f(n')$.*

Case ($\Rightarrow$)

Consider the subgraph of PG reachable from m and m'. From $m \leq m'$ and the definition of ($\leq$), we know that $m' \geq_{rank} m$.

From Lemma 7, and $m' \geq_{rank} m$ we know that $f_{pg}(m') \geq_{rank} f_{pg}(m)$. From Lemma 9, there is a transition $f(n) \xrightarrow{f(l)} f(n')$.

Case ($\Leftarrow$)

Let $M = f(m)$ and let $M' = f(m')$. Suppose that $M \leq M'$, which means that with $M' \geq_{rank} M$ in the abstract product graph. It also means that, for each neighbor N', if we have the transition: $M' \xrightarrow{L} N'$ then we also have a transition: $M \xrightarrow{L} N$ where $N' \geq_{rank} N$. We must show that the same relation holds for m and m'. Assume that $\text{topo}(M) = f(a)$ and $\text{topo}(N) = f(b)$.

Also assume that m' has a neighbor n'. We know that $\text{topo}(m) = a$ *and* $\text{topo}(m') = b$*. Clearly* (a, b) *is a valid concrete topology edge since it is a transition in the product graph between* m *and* m'*. We now apply Lemma 8 with the following facts:*

1. (a, b) *is a concrete topology edge*
2. $\text{topo}(m) = a$
3. $f_{pg}(m) = M$
4. (M, N) *is an edge in* PG^A
5. $\text{topo}(N) = f(b)$

Lemma 8 lets us conclude that there exists a node n *such that:*

1. $\text{topo}(n) = b$
2. $f_{pg}(n) = N$
3. (m, n) *is a concrete edge in* PG
4. $\text{rank}(n) = \text{rank}(N)$

Thus if there is transition $m' \xrightarrow{b} n'$*, then there is also a transition* $m \xrightarrow{b} n$*. Lemma 7 together with the assumption* $N' \geq_{rank} N$ *and the fact that* $\text{rank}(n) = \text{rank}(N)$ *lets us conclude that* $n' \geq_{rank} n$*.*

□

Note: Lemma 10 tells us that the total ordering between node preferences is preserved under the lifted graph homomorphism f_{pg}. We use this fact to define the local preference function ($\text{pref} : V \to \mathbb{N}$) that maps product graph nodes to numbers reflecting the total ordering such that $\text{pref}(m) = \text{pref}(f(m))$ for all product graph nodes m. We do this to normalize policies so they refer to the exact same BGP local preferences. This is useful because we are proving syntactic equivalence of commutativity rather than semantic equivalence.

Lemma A.2.11. *For all nodes* m*,* $\text{pref}(m) = \text{pref}^A(f_{pg}(m))$.

Proof: pref *and* pref^A *assign values to PG nodes for each topology location as increasing integers starting from 0 according to the total ordering for* $\leq$. *We know that* pref *and* pref^A *preserve the total ordering since: by construction* $\text{pref}(m) \leq \text{pref}(m') \iff m \leq m'$ *and* $\text{pref}(f_{pg}(m)) \leq \text{pref}(f_{pg}(m')) \iff f_{pg}(m) \leq f_{pg}(m')$, *and from Lemma 10 we know that* $m \leq m' \iff f_{pg}(m) \leq f_{pg}(m')$. *Also, because there can only be a single product graph node for a particular location and set of states (since automata are determinized), and from Lemma 5 we know that for each* m *there exists a node* $f(m)$ *with the same states it must be the case that* $\text{pref}(m) = \text{pref}^A(f_{pg}(m))$ □

Constraint Substitution (Part 2)

In this section, we aim to prove that concretization and compilation commute for individual Propane/AT constraints. Since a policy is simply an ordered sequence of constraints, it is then straightforward to show that compilation and concretization commute for entire policies. In particular, we are interested in showing that for all constraints p, contexts Γ, and graph homomorphisms $f : G \to G^A$ between G and G^A (without multiplicities):

$$\text{con}(\text{compile}(p, G^A), \Gamma, f, G) = \text{compile}(\text{con}(p, \Gamma, f), G)$$

The proof goes by case analysis on the test t in the constraint p, and makes heavy use of the lemmas about the product graphs from the previous section. We first use one helper lemma here that relates sets of neighbors in the concrete and abstract product graphs under the inverse homomorphism f^{-1}:

Lemma A.2.12. *For concrete* (PG) *and abstract* (PG^A) *product graphs related by* f_{pg}, *if* $f(\ell) = \text{topo}(M)$, *then*

$$
\begin{array}{l}
[\, pfx \rightarrow [\, ma \mid \\
\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad pin = \mathrm{adjIn}(PG^A, M), \\
\quad (in, q_N) \in \{(BS, q_N) \mid \\
\qquad\qquad BS = \{b \mid b \in f^{-1}(B),\ (b, \ell) \in G.E,\ (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad out \leftarrow \{c \mid c \in f^{-1}(C), (c, \ell) \in G.E, (C, \_) \in \mathrm{adjOut}(PG^A, M)\}, \\
\quad ma = ord^A(M) : (in, q_N) \rightarrow (out, q_M)\,]\,]
\end{array}
$$

Is equivalent to

$$
\begin{array}{l}
[\, pfx \rightarrow [\, ma \mid \\
\quad m \leftarrow (l, q_m) \in PG, \\
\quad pin \leftarrow \mathrm{adjIn}(PG, m), \\
\quad (in, q_n) \leftarrow \{(bs, q_n) \mid bs = \{b \mid (b, q_n) \in pin\}, bs \neq \emptyset\}), \\
\quad out \leftarrow \{c \mid (c, \_) \in \mathrm{adjOut}(PG, m)\}, \\
\quad ma = ord^A(f_{pg}(m)) : (in, q_n) \rightarrow (out, q_m)\,]\,]
\end{array}
$$

Proof:

Suppose for node M, there exists a node $m = (l, q_m)$ such that $f_{pg}(m) = M$. We want to show the following:

$$\{b \mid b \in f^{-1}(B),\ (b, \ell) \in G.E,\ (B, q_N) \in \mathrm{adjIn}(PG^A, f_{pg}(m))\}$$

is equivalent to

$$\{b \mid (b, q_N) \in \mathrm{adjIn}(PG, m)\}$$

Suppose that an element $x \in \{b \mid (b, q_N) \in \mathrm{adjIn}(PG, m)\}$. Then $(b, q_N) \in \mathrm{adjIn}(PG, m)$ and there is an edge (b, ℓ) in the topology between b and $\mathrm{topo}(m)$. From the homomorphism, this means

$(f(b), q_N) \in \text{adjIn}(PG^A, f(m))$. *It follows that* $x \in \{b \mid b \in f^{-1}(B),\ (b, \ell) \in G.E,\ (B, q_N) \in \text{adjIn}(PG^A, f_{pg}(m))\}$.

Now suppose the other direction. There is an element $x \in \{b \mid b \in f^{-1}(B),\ (b, \ell) \in G.E,\ (B, q_N) \in \text{adjIn}(PG^A, f_{pg}(m))\}$. *We know that* $f(x) = B$, *and that there is a topology edge between* b *and* ℓ. *Lemma 8 together with the fact that* $f(\ell) = \text{topo}(m)$ *lets us conclude that there is an edge in* PG *between* (b, q_N) *and* m. *It follows then that* $x \in \{b \mid (b, q_N) \in \text{adjIn}(PG, m)\}$.

The same reasoning can be applied to adjOut as well. Further, note that because we are looking at a particular location l, *there can only be at most one such* $m = (l, q_N)$ *such that* $f_{pg}(m) = M = (f(l), q_N)$ *since the automata are deterministic. This means that* $M \leftarrow (l, q_M) \in PG^A$ *can be replaced with* $m \leftarrow (l, q_m) \in PG$ *since for each* M *we will either have some* m *such that* $f_{pg}(m) = M$ *and the above equivalence holds, or else there is no such* M, *in which case the set is filtered by the condition:* $BS \neq \emptyset$.

□

Theorem A.2.13. *For all Propane/AT constraints* p, *contexts* Γ, *topologies* G *and* G^A *related by homomorphism* $f : G \to G^A$*:*

$$\text{con}(\text{compile}(p, G^A), \Gamma, f, G) = \text{compile}(\text{con}(p, \Gamma, f), G)$$

Proof: *From the grammar, we know that* $p = (t\text{=>}\ r_1\ \text{>>}\ \ldots\ \text{>>}\ r_n)$. *The proof proceeds by cases on* t. *In each case, we expand the definition of* compile *and* con *until they no longer appear in the term. We then appeal to graph properties of* PG *and* PG^A *to rewrite the terms into equivalent forms.*

Case *(*$t = pfx$*)*

For the right side:

$$
\begin{aligned}
& \mathrm{compile}(\mathrm{con}(p, \Gamma, f), G) \\
= \; & \mathrm{compile}(\mathrm{con}(pfx{=>}\, r_1 \,{>>}\, \ldots, \Gamma, f), G) && \textit{Substitution} \\
= \; & \mathrm{compile}(pfx{=>}\, \mathrm{con}(r_1, \Gamma, f) \,{>>}\, \ldots, G) && \textit{Definition} \\
= \; & \mathrm{compile}(pfx{=>}\, \mathrm{con}(r_1, \Gamma, f) \cap \mathrm{end}(\Sigma) \,{>>}\, \ldots, G) && \textit{Regex Equiv.} \\
= \; & \mathrm{compile}_{\mathrm{mBGP}}([\mathrm{compile}_{\mathrm{PG}}(pfx{=>} && \textit{Definition} \\
& \qquad \mathrm{con}(r_1, \Gamma, f) \cap \mathrm{end}(\Sigma) \,{>>}\, \ldots, G)]) \\
= \; & \mathrm{compile}_{\mathrm{mBGP}}([(pfx, PG, \mathrm{pref})]) && \textit{Assumption}
\end{aligned}
$$

Similarly, for the left side:

$$
\begin{aligned}
& \mathrm{con}(\mathrm{compile}(p, G^A), \Gamma, f, G) \\
= \; & \mathrm{con}(\mathrm{compile}(pfx{=>}\, r_1 \,{>>}\, \ldots), \Gamma, f, G) && \textit{Substitution} \\
= \; & \mathrm{con}(\mathrm{compile}_{\mathrm{mBGP}}([\mathrm{compile}_{\mathrm{PG}}(pfx{=>}\, r_1 \,{>>}\, \ldots)]), \Gamma, f, G) && \textit{Definition} \\
= \; & \mathrm{con}(\mathrm{compile}_{\mathrm{mBGP}}([(pfx, PG^A, \mathrm{pref}^A)]), \Gamma, f, G) && \textit{Assumption}
\end{aligned}
$$

Both sides of the equality are now in a form where we can apply the lemmas from the previous section. This allows us to relate PG and PG^A as well as pref *and* pref^A*. Continuing to expand the right side by substituting the definition of* $\mathrm{compile}_{\mathrm{mBGP}}$ *we get:*

(Right-hand side)

$$
\begin{array}{l}
[\, l \to rc \mid \\
\quad l \in internal(G.V),\ rc = \\
\quad\quad [\, pfx \to [\, ma \mid \\
\quad\quad\quad m \leftarrow (l, q_m) \in PG, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG, m), \\
\quad\quad\quad (in, q_n) \leftarrow \{(bs, q_n) \mid bs = \{b \mid (b, q_n) \in pin\}, bs \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{c \mid (c, \_) \in \textbf{\textit{adjOut}}(PG_i, m)\}, \\
\quad\quad\quad ma = \mathrm{pref}(m) : (in, q_n) \to (out, q_m)\,]\,]\,]
\end{array}
$$

Next we further expand on the left side for the abstract topology to show it is equivalent:

(Left-hand side)

$$
\begin{array}{l}
\mathrm{con}([\, l \to rc \mid \\
\quad l \in internal(G^A.V),\ rc = \\
\quad\quad [\, pfx \to [\, ma \mid \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N) \mid BS = \{B \mid (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{C \mid (C, \_) \in \textbf{\textit{adjOut}}(PG^A, M)\}, \\
\quad\quad\quad ma = \mathrm{pref}^A(M) : (in, q_N) \to (out, q_M)\,]\,]\,], \Gamma, f, G)
\end{array}
$$

We apply the definition of con *to push it inside the list:*

$$
\begin{array}{l}
[\,\ell \to \mathrm{con}(rc, \ell, \Gamma, f, G) \mid \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = \\
\quad\quad [\,pfx \to [\,ma \mid \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textit{adjIn}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N) \mid BS = \{B \mid (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{C \mid (C, \_) \in \textit{adjOut}(PG^A, M)\}, \\
\quad\quad\quad ma = \mathrm{pref}^A(M) : (in, q_N) \to (out, q_M)\,]\,]\,]
\end{array}
$$

We now apply the definition of con *to push it further inside:*

$$
\begin{array}{l}
[\,\ell \to rc \mid \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = \\
\quad\quad \mathrm{con}([\,pfx \to [\,ma \mid \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textit{adjIn}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N) \mid BS = \{B \mid (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{C \mid (C, \_) \in \textit{adjOut}(PG^A, M)\}, \\
\quad\quad\quad ma = \mathrm{pref}^A(M) : (in, q_N) \to (out, q_M)\,]\,], \Gamma, f, G)\,]
\end{array}
$$

Once again, we apply the definition of con

$$
\begin{array}{l}
[\ \ell \to rc \mid \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = \\
\quad\quad [\ pfx \to [\ ma \mid \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N) \mid BS = \{B \mid (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{C \mid (C, \_) \in \textbf{\textit{adjOut}}(PG^A, M)\}, \\
\quad\quad\quad ma = \mathrm{con}(\mathrm{pref}^A(M) : (in, q_N) \to (out, q_M), true, \ell, \Gamma, f, G)\ ]\ ]\ ]
\end{array}
$$

Next, we rewrite the sets in *and* out *according to* con*:*

$$
\begin{array}{l}
[\ \ell \to rc \mid \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = \\
\quad\quad [\ pfx \to [\ ma \mid \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG^A, M), \\
\quad\quad\quad (\mathrm{con}(in, \ell, \Gamma, f, G), q_N) \leftarrow \\
\quad\quad\quad\quad\quad\quad \{(BS, q_N) \mid BS = \{B \mid (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \mathrm{con}(\{C \mid (C, \_) \in \textbf{\textit{adjOut}}(PG^A, M)\}, \ell, \Gamma, f, G), \\
\quad\quad\quad ma = \mathrm{pref}^A(M) : (in, q_N) \to (out, q_M)\ ]\ ]\ ]
\end{array}
$$

Rewrite con *in the set* in*:*

$$
\begin{array}{l}
[\,\ell \to rc \mid \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = \\
\quad\quad [\,pfx \to [\,ma \mid \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N) \mid \\
\quad\quad\quad\quad\quad BS = \mathrm{con}(\{B \mid (B, q_N) \in pin\}, \ell, \Gamma, f, G), BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \mathrm{con}(\{C \mid (C, \_) \in \textbf{\textit{adjOut}}(PG^A, M)\}, \ell, \Gamma, f, G), \\
\quad\quad\quad ma = \mathrm{pref}^A(M) : (in, q_N) \to (out, q_M)\,]\,]\,]
\end{array}
$$

Apply the definition of con *on the inner set of* in *and* out*:*

$$
\begin{array}{l}
[\,\ell \to rc \mid \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = \\
\quad\quad [\,pfx \to [\,ma \mid \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N) \mid \\
\quad\quad\quad\quad BS = \{b \mid b \in f^{-1}(B), (b, \ell) \in G.E, (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{c \mid c \in f^{-1}(C), (c, \ell) \in G.E, (C, \_) \in \textbf{\textit{adjOut}}(PG^A, M)\}, \\
\quad\quad\quad ma = \mathrm{con}(\mathrm{pref}^A(M) : (in, q_N) \to (out, q_M), true, \ell, \Gamma, f, G)\,]\,]\,]
\end{array}
$$

Applying Lemma 12 with the fact that $f(\ell) = l = \mathrm{topo}(M)$ *this is equivalent to:*

$$
\begin{array}{l}
[\, l \to rc \mid \\
\quad l \in internal(G.V),\ rc = \\
\quad\quad [\, pfx \to [\, ma \mid \\
\quad\quad\quad m \leftarrow (l, q_m) \in PG, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG, m), \\
\quad\quad\quad (in, q_n) \leftarrow \{(bs, q_n) \mid bs = \{b \mid (b, q_n) \in pin\}, bs \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{c \mid (c, \_) \in \textbf{\textit{adjOut}}(PG, m)\}, \\
\quad\quad\quad ma = \mathrm{pref}^A(f_{pg}(m)) : (in, q_) \to (out, q_m)\,]\,]\,]
\end{array}
$$

From Definition 2, we know that the $\mathrm{pref}(m) = \mathrm{pref}(f_{pg}(m))$*:*

$$
\begin{array}{l}
[\, l \to rc \mid \\
\quad l \in internal(G.V),\ rc = \\
\quad\quad [\, pfx \to [\, ma \mid \\
\quad\quad\quad m \leftarrow (l, q_m) \in PG, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG, m), \\
\quad\quad\quad (in, q_n) \leftarrow \{(bs, q_n) \mid bs = \{b \mid (b, q_n) \in pin\}, bs \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{c \mid (c, \_) \in \textbf{\textit{adjOut}}(PG, m)\}, \\
\quad\quad\quad ma = \mathrm{pref}(m) : (in, q_n) \to (out, q_m)\,]\,]\,]
\end{array}
$$

This is exactly the result we obtained from expanding the right hand side of the equality.

Case *($t = \$x$)*

The case for $t = \$x$ follows a similar line of reasoning. We start by evaluating the right-hand side:

(Right-hand side)

$$
\begin{aligned}
& \mathrm{compile}(\mathrm{con}(p, \Gamma, f), G) \\
= \;& \mathrm{compile}(\mathrm{con}(\$x \texttt{=>}\ r_1 \texttt{>>}\ \ldots, \Gamma, f), G) && \textit{Substitution} \\
= \;& \mathrm{compile}([pfx \texttt{=>}\ \mathrm{con}(r_1, \Gamma, f) \cap \mathrm{end}(l) \texttt{>>}\ \ldots \mid && \textit{Definition of con} \\
& \qquad (pfx, l) \in \Gamma(x)], G) \\
= \;& \mathrm{compile}_{\mathrm{mBGP}}([\mathrm{compile}_{\mathrm{PG}}(pfx \texttt{=>} && \textit{Definition of compile} \\
& \qquad \mathrm{con}(r_1, \Gamma, f) \cap \mathrm{end}(l) \texttt{>>} \\
& \qquad \ldots \texttt{>>} \\
& \qquad \mathrm{con}(r_n, \Gamma, f) \cap \mathrm{end}(l), G) \mid \\
& \quad (pfx, l) \in \Gamma(x)]) \\
= \;& \mathrm{compile}_{\mathrm{mBGP}}([(pfx_1, PG_1, \mathrm{pref}_1), \ldots, (pfx_k, PG_k, \mathrm{pref}_k)]) && \textit{Assumption}
\end{aligned}
$$

Expanding further, this becomes:

$$
\begin{aligned}
& [\, l \to rc \mid \\
& \quad l \in internal(G.V),\ rc = append_i \\
& \quad [\, pfx_i \to [\, ma \mid \\
& \qquad (pfx_i, \_) \in \Gamma(x), \\
& \qquad m \leftarrow (l, q_m) \in PG_i, \\
& \qquad pin \leftarrow \textbf{\textit{adjIn}}(PG_i, m), \\
& \qquad (in, q_n) \leftarrow \{(bs, q_n) \mid bs = \{b \mid (b, q_n) \in pin\}, bs \neq \emptyset\}, \\
& \qquad out \leftarrow \{c \mid (c, \_) \in \textbf{\textit{adjOut}}(PG_i, m)\}, \\
& \qquad ma = \mathrm{pref}_i(m) : (in, q_n) \to (out, q_m)\,]\,]\,]
\end{aligned}
$$

As before, we now show that the left hand side is equivalent by applying the definitions of con *and* compile*:*

(Left-hand side)

$$
\begin{aligned}
& \mathrm{con}(\mathrm{compile}(p, G^A), \Gamma, f, G) \\
= \;& \mathrm{con}(\mathrm{compile}(\$x \texttt{=>}\ r_1 \texttt{>>}\ \ldots, G^A), \Gamma, f, G) && \textit{Substitution} \\
= \;& \mathrm{con}(\mathrm{compile}_{\mathrm{mBGP}}([\mathrm{compile}_{\mathrm{PG}}(\$x \texttt{=>}\ r_1 \texttt{>>}\ \ldots, G^A)]), \Gamma, f, G) && \textit{Definition} \\
= \;& \mathrm{con}(\mathrm{compile}_{\mathrm{mBGP}}([(\$x, PG^A, \mathrm{pref}^A)]), \Gamma, f, G) && \textit{Assumption}
\end{aligned}
$$

Note that PG_i and PG^A, as well as pref *and* pref^A *are related by the lemmas from the previous section. Next we further expand on the left side for the abstract topology to show it is equivalent:*

$$
\begin{aligned}
& \mathrm{con}([\, l \to rc \mid \\
& \quad l \in internal(G^A.V),\ rc = \\
& \quad\quad [\, \$x \to [\, ma \mid \\
& \quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
& \quad\quad\quad pin \leftarrow \textit{adjIn}(PG^A, M), \\
& \quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N) \mid BS = \{B \mid (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
& \quad\quad\quad out \leftarrow \{C \mid (C, \_) \in \textit{adjOut}(PG^A, M)\}, \\
& \quad\quad\quad ma = \mathrm{pref}^A(M) : (in, q_N) \to (out, q_M)\ ]\ ]\ ], \Gamma, f, G)
\end{aligned}
$$

We apply the definition of con *to push it inside the list:*

$$
\begin{array}{l}
[\ \ell \to \mathrm{con}(rc, \ell, \Gamma, f, G)\ | \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = \\
\quad\quad [\ \$x \to [\ ma\ | \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textit{adjIn}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N)\ |\ BS = \{B\ |\ (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{C\ |\ (C, \_) \in \textit{adjOut}(PG^A, M)\}, \\
\quad\quad\quad ma = \mathrm{pref}^A(M) : (in, q_N) \to (out, q_M)\ ]\ ]\ ]
\end{array}
$$

We now apply the definition of con *to push it further inside:*

$$
\begin{array}{l}
[\ \ell \to rc\ | \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = \\
\quad\quad \mathrm{con}([\ \$x \to [\ ma\ | \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textit{adjIn}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N)\ |\ BS = \{B\ |\ (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{C\ |\ (C, \_) \in \textit{adjOut}(PG^A, M)\}, \\
\quad\quad\quad ma = \mathrm{pref}^A(M) : (in, q_N) \to (out, q_M)\ ]\ ], \Gamma, f, G)\ ]
\end{array}
$$

Once again, we apply the definition of con

$$
\begin{array}{l}
[\,\ell \to rc \mid \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = append_i \\
\quad\quad [\,pfx_i \to [\,ma \mid \\
\quad\quad\quad (pfx_i, l') \in \Gamma(x), \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N) \mid BS = \{B \mid (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{C \mid (C, \_) \in \textbf{\textit{adjOut}}(PG^A, M)\}, \\
\quad\quad\quad ma = \mathrm{con}(\mathrm{pref}^A(M) : (in, q_N) \to (out, q_M), \Gamma, f, G)\,]\,]\,]
\end{array}
$$

Once again, we apply the definition of con

$$
\begin{array}{l}
[\,\ell \to rc \mid \\
\quad \ell \in f^{-1}(l), \\
\quad l \in internal(G^A.V),\ rc = append_i \\
\quad\quad [\,pfx_i \to [\,ma \mid \\
\quad\quad\quad (pfx_i, l') \in \Gamma(x), \\
\quad\quad\quad M \leftarrow (l, q_M) \in PG^A, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG^A, M), \\
\quad\quad\quad (in, q_N) \leftarrow \{(BS, q_N) \mid BS = \{B \mid (B, q_N) \in pin\}, BS \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{C \mid (C, \_) \in \textbf{\textit{adjOut}}(PG^A, M)\}, \\
\quad\quad\quad ma = \textbf{\textit{if}}\ in = \{G.start\}\ \textbf{\textit{and}}\ l' = \ell\ \textbf{\textit{then}}\ \bullet \\
\quad\quad\quad\quad\quad \textbf{\textit{else}}\ \mathrm{pref}^A(M) : (\mathrm{con}(in, \ell, \Gamma, f, G), q_N) \to \\
\quad\quad\quad\quad\quad\quad\quad (\mathrm{con}(out, \ell, \Gamma, f, G), q_M)\,]\,]\,]
\end{array}
$$

After filtering the cases where $in = \{G.start\}$ *and* $l' = \ell$*, we have the same situation as the previous case. However, we know that each* PG_i *was constructed from* $con(r_i, \ell, \Gamma, f, G) \cap end(l_i)$*.*

This means that the only state in PG_i connected to the start node ($start$) is a node for l_i. This was ensured by the fact that the transition function for each regular expression $\sigma_i(q_0, l) = q$ is defined only for locations $l \in L$ where $L = \{l_i\}$ from the construction of the automata. We apply Lemma 12 with the fact that $f(\ell) = \text{topo}(M)$:

$$
\begin{array}{l}
[\, \ell \to rc \mid \\
\quad l \in internal(G.V),\ rc = append_i \\
\quad\quad [\, pfx_i \to [\, ma \mid \\
\quad\quad\quad (pfx_i, l') \in \Gamma(x), \\
\quad\quad\quad m \leftarrow (l, q_m) \in PG_i, \\
\quad\quad\quad pin \leftarrow \textbf{\textit{adjIn}}(PG_i, m), \\
\quad\quad\quad (in, q_n) \leftarrow \{(bs, q_n) \mid bs = \{b \mid (b, q_n) \in pin\}, bs \neq \emptyset\}, \\
\quad\quad\quad out \leftarrow \{C \mid (C, \_) \in \textbf{\textit{adjOut}}(PG_i, m)\}, \\
\quad\quad\quad ma = \text{pref}_i(m) : (in, q_n) \to (out, q_m)\,]\,]\,]
\end{array}
$$

□

Substitution of Policies (Step 3)

The final step is now to show the commutativity property for entire Propane/AT policies.

Theorem A.2.14. *For all Propane/AT policies pol, contexts Γ, topologies G and G^A related by homomorphism $f : G \to G^A$:*

$$\text{con}(\text{compile}(pol, G^A), \Gamma, f, G) = \text{compile}(\text{con}(pol, \Gamma, f), G)$$

Proof:

The proof simply uses Theorem 13 after rewriting the policy according to con *and* compile*. Given a policy* $pol = p_1, \ldots, p_n$*, we compute the left-hand side:*

$$
\begin{array}{lll}
 & \mathrm{con}(\mathrm{compile}(\mathit{pol}, G^A), \Gamma, f, G) & \\
= & \mathrm{con}(\mathrm{compile}(p_1, \ldots, p_n, G^A), \Gamma, f, G) & \textit{Substitution} \\
= & \mathrm{con}(\mathrm{compile}_{\mathrm{mBGP}}([& \textit{Definition of Compile} \\
 & \quad \mathrm{compile}_{\mathrm{PG}}(p_1, G^A), & \\
 & \quad \ldots, & \\
 & \quad \mathrm{compile}_{\mathrm{PG}}(p_n, G^A)]), \Gamma, f, G) & \\
= & \mathrm{con}(\mathrm{compile}_{\mathrm{mBGP}}([& \textit{Assumption} \\
 & \quad (t_1, PG_1^A, \mathrm{pref}_1^A), & \\
 & \quad \ldots, & \\
 & \quad (t_n, PG_n^A, \mathrm{pref}_n^A)]), \Gamma, f, G) &
\end{array}
$$

For the right side of the equation:

$$
\begin{array}{lll}
 & \mathrm{compile}(\mathrm{con}(\mathit{pol}, \Gamma, f), G) & \\
= & \mathrm{compile}(\mathrm{con}(p1, \ldots, p_n, \Gamma, f), G) & \textit{Substitution} \\
= & \mathrm{compile}(\mathrm{con}(p1, \Gamma, f), \ldots, \mathrm{con}(p_n, \Gamma, f), G) & \textit{Definition of con} \\
= & \mathrm{compile}_{\mathrm{mBGP}}([& \textit{Definition of compile} \\
 & \quad \mathrm{compile}_{\mathrm{PG}}(\mathrm{con}(p1, \Gamma, f), G), & \\
 & \quad \ldots, & \\
 & \quad \mathrm{compile}_{\mathrm{PG}}(\mathrm{con}(p_n, \Gamma, f), G)]) & \\
= & \mathrm{compile}_{\mathrm{mBGP}}([(t'_1, PG_1, \mathrm{pref}_1), \ldots, (t'_n, PG_n, \mathrm{pref}_n)]) & \textit{Assumption}
\end{array}
$$

(Given)

First, we observe that, for each p_i*, Theorem 11 and* $\mathrm{compile}_{\mathrm{PG}}$ *give us that, for each* i*:*

$$
\begin{aligned}
& \mathrm{con}([\, l \to rc \mid l \in internal(G^A.V), && \textit{Left side} \\
& \qquad rc = [\mathrm{compile}_{\mathrm{pred}}((t_i, PG_i^A, \mathrm{pref}_i^A))]\ ], \Gamma, f, G) \\
= \ & [\, l \to rc \mid l \in internal(G.V), && \textit{Definition of con} \\
& \qquad rc = [\mathrm{con}(\mathrm{compile}_{\mathrm{pred}}((t_i, PG_i^A, \mathrm{pref}_i^A)), \Gamma, f, G)]\ ] \\
= \ & [\, l \to rc \mid l \in internal(G.V), && \textit{Right side} \\
& \qquad rc = [\mathrm{compile}_{\mathrm{pred}}((t'_i, PG_i, \mathrm{pref}_i))]\ ]
\end{aligned}
$$

(To Show)

By applying the definition of $\mathrm{compile}_{\mathrm{mBGP}}$*, we start with the left hand side:*

$$
\begin{aligned}
& \mathrm{con}([\, l \to rc \mid \\
& \qquad l \in internal(G^A.V), \\
& \qquad rc = append_{i \in \{1..n\}}(\mathrm{compile}_{\mathrm{pred}}((t_i, PG_i^A, \mathrm{pref}_i^A)))\ ], \Gamma, f, G)
\end{aligned}
$$

By applying the definition of con *to the left-hand side, we get:*

$$
\begin{aligned}
& [\, l \to rc \mid \\
& \qquad l \in internal(G.V), \\
& \qquad rc = \mathrm{con}(append_{i \in \{1..n\}}(\mathrm{compile}_{\mathrm{pred}}((t_i, PG_i^A, \mathrm{pref}_i^A))), \Gamma, f, G)\ ]
\end{aligned}
$$

Applying the definition of con *one more time:*

$$
\begin{aligned}
& [\, l \to rc \mid \\
& \qquad l \in internal(G.V), \\
& \qquad rc = append_{i \in \{1..n\}} \mathrm{con}((\mathrm{compile}_{\mathrm{pred}}((t_i, PG_i^A, \mathrm{pref}_i^A)), \Gamma, f, G))\ ]
\end{aligned}
$$

From the definition of list append *and the given equality from Theorem 13 shown above, we obtain the right hand side.*

$$
\begin{aligned}
&[\, l \to rc \mid \\
&\quad l \in internal(G.V), \\
&\quad rc = append_{i \in \{1..n\}}(\mathrm{compile}_{\mathrm{pred}}((t'_i, PG_i, \mathrm{pref}_i)))\,]
\end{aligned}
$$

□

Bibliography

[1] Cisco ios technologies. https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-technologies/index.html, 2018.

[2] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. Hyperx: Topology, routing, and packaging of efficient large-scale networks. In *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, August 2008.

[4] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing policy specification language (RPSL). RFC 2622, RFC Editor, June 1999. http://www.rfc-editor.org/rfc/rfc2622.txt.

[5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *POPL*, January 2014.

[6] M Anderson. Time warner cable says outages largely resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved, August 2014.

[7] Alexey Andreyev. Introducing data center fabric, the next-generation facebook data center network. https://code.facebook.com/posts/360346274145943/, 2014.

[8] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[9] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Trans. Information and System Security*, 14(1), 2011.

[10] Ryan Beckett. Propane compiler. https://github.com/rabeckett/propane, 2016.

[11] Ryan Beckett. Minesweeper source code. https://batfish.github.io/minesweeper, 2017.

[12] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. August 2017.

[13] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. August 2018.

[14] 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 *SIGCOMM*, 2016.

[15] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitu Padhye, and David Walker. Network configuration synthesis with abstract topologies. June 2017.

[16] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an open, distributed SDN OS. In *HotSDN*, August 2014.

[17] News and press — BGPMon. `http://www.bgpmon.net/news-and-events/`.

[18] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.

[19] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, September 1983.

[20] M. Bjorklund. YANG - a data modeling language for the network configuration protocol (NETCONF). RFC 6020, RFC Editor, October 2010. `http://www.rfc-editor.org/rfc/rfc6020.txt`.

[21] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. *νZ - An Optimizing SMT Solver*. 2015.

[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[23] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[24] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, August 2007.

[25] Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, 2010.

[26] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 8(2), 1986.

[27] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *Computer Aided Verification, 5th International Conference, CAV, Proceedings*, pages 450–462, 1993.

[28] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV, Proceedings*, pages 154–169, 2000.

[29] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[30] Wikipedia contributors. Content-addressable memory — Wikipedia, the free encyclopedia. [Online; accessed March-2018].

[31] Wikipedia contributors. ping (networking utility) — Wikipedia, the free encyclopedia. [Online; accessed March-2018].

[32] Wikipedia contributors. Traceroute — Wikipedia, the free encyclopedia. [Online; accessed March-2018].

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[34] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. POPL '77, pages 238–252, January 1977.

[35] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[36] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9), 2011.

[37] Steve Dent. Comcast's nationwide outage was caused by a configuration error. `https://www.engadget.com/2017/11/07/comcast-internet-outage-level-3-route-leak/`, 2017.

[38] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In *Computer Aided Verification, 5th International Conference, CAV, Proceedings*, pages 463–478, 1993.

[39] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network configuration protocol (NETCONF). RFC 6241, RFC Editor, June 2011. `http://www.rfc-editor.org/rfc/rfc6241.txt`.

[40] 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 *OSDI*, 2016.

[41] Nick Feamster and Hari Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, May 2005.

[42] Ashley Flavel and Matthew Roughan. Stable and flexible ibgp. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 183–194, October 2009.

[43] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, 1967.

[44] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *NSDI*, March 2015.

[45] Nate Foster, Michael J. Freedman, Arjun Guha, Rob Harrison, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Mark Reitblatt, Jennifer Rexford, Cole Schlesinger, Alec Story, and David Walker. Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134, February 2013.

[46] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, pages 282–309, 2016.

[47] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. In *SIGMETRICS*, 2000.

[48] Wouter Gelade and Frank Neven. Succinctness of the complement and intersection of regular expressions. *ACM Trans. Comput. Logic*, 13(1):4:1–4:19, January 2012.

[49] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, August 2016.

[50] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. Management plane analytics. In *Internet Measurement Conference (IMC)*, 2015.

[51] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, August 2011.

[52] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[53] Joanne Godfrey. The summer of network misconfigurations. `https://blog.algosec.com/2016/08/business-outages-caused-misconfigurations-headline-news-summer.html`, 2016.

[54] 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 *SIGCOMM*, 2009.

[55] Tim Griffin. BGP wedgies. https://www.ietf.org/rfc/rfc4264.txt, 2005.

[56] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Networking*, 10(2), 2002.

[57] Timothy G. Griffin and João Luís Sobrinho. Metarouting. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 1–12, August 2005.

[58] Timothy G. Griffin and Gordon Wilfong. On the correctness of IBGP configuration. In *SIGCOMM*, August 2002.

[59] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[60] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[61] Hatch – create and share configurations. http://www.hatchconfigs.com/.

[62] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[63] Don Jacob. The cost and cause of network downtime. https://www.packetdesign.com/blog/cost-and-cause-of-network-downtime/, May 2016.

[64] Alex Johnson and Jay Blackman. United airlines domestic flights grounded for 2 hours by computer outage. https://www.nbcnews.com/storyline/airplane-mode/all-united-airlines-domestic-flights-grounded-computer-outage-n710596, 2017.

[65] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[66] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 10:1–10:12, 2016.

[67] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[68] Zeus Kerravala. What is behind network downtime? proactive steps to reduce human error and improve availability of networks. https://www.cs.princeton.edu/courses/archive/fall10/cos561/papers/Yankee04.pdf, 2004.

[69] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[70] John Kim, William J. Dally, and Dennis Abts. Flattened butterfly: A cost-efficient topology for high-radix networks. In *ISCA*, 2007.

[71] P. Lapukhov, A. Premji, and J. Mitchell. Use of BGP for routing in large-scale data centers. Internet draft, August 2015.

[72] Franck Le, Geoffrey Xie, and Hui Zhang. Understanding route redistribution. In *ICNP*, 2007.

[73] Franck Le, Geoffrey G. Xie, and Hui Zhang. On route aggregation. In *CoNEXT*, December 2011.

[74] Thomas Lengauer and Robert Tarjan. A fast algorithm for finding dominators in a flowgraph. In *TOPLAS*, July 1979.

[75] Leonid Libkin and Domagoj Vrgoč. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 74–85, 2012.

[76] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *NSDI*, 2013.

[77] Nuno P. Lopes, Nikolaj Bjorner, and Patrice Godefroid. Network verification in the light of program verification. In *Technical Report*, 2013.

[78] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.

[79] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8), 2009.

[80] James McCauley, Aurojit Panda, Martin Casado, Teemu Koponen, and Scott Shenker. Extending SDN to large-scale networks. In *Open Networking Summit*, April 2013.

[81] Sanjai Narain. Network configuration management via model finding. In *LISA*, December 2005.

[82] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network Systems Management*, 16(3):235–258, October 2008.

[83] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. In *J. Funct. Program.*, March 2009.

[84] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. In *POPL*, 2016.

[85] Quagga routing suite. `http://www.nongnu.org/quagga/`.

[86] Mark Reitblatt, Marco Canini, Nate Foster, and Arjun Guha. FatTire: Declarative fault tolerance for software defined networks. In *HotSDN*, August 2013.

[87] Press Release. New algosec survey reveals lack of security automation exposes enterprises to cyber attacks and outages. https://www.algosec.com/press_release/new-algosec-survey-reveals-lack-security-automation-exposes-enterprises-cyber-attacks-outages/, 2016.

[88] Public Safety and Homeland Security Bureau. Level 3 nationwide outage. https://transition.fcc.gov/Daily_Releases/Daily_Business/2018/db0313/DOC-349661A1.pdf, 2018.

[89] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. Condor: Better topologies through declarative design. In *SIGCOMM*, 2015.

[90] Simon Sharwood. Google cloud wobbles as workers patch wrong routers. http://www.theregister.co.uk/2016/03/01/google_cloud_wobbles_as_workers_patch_wrong_routers/, 2016.

[91] Ryan Singel. Pakistan's accidental youtube re-routing exposes trust flaw in net. https://www.wired.com/2008/02/pakistans-accid/, 2008.

[92] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3), 1985.

[93] João 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*, SIGCOMM '03, pages 49–60, August 2003.

[94] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *CoNEXT*, December 2014.

[95] Yevgenly Sverdlik. Microsoft: misconfigured network device led to azure outage. http://www.datacenterdynamics.com/content-tracks/servers-storage/microsoft-misconfigured-network-device-led-to-azure-outage/68312.fullarticle, 2012.

[96] Robert Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. 18:110–127, 04 1979.

[97] configuration templates — thwack. https://thwack.solarwinds.com/search.jspa?q=configuration+templates.

[98] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *In Tools and Algorithms for Construction and Analysis of Systems (TACAS*, pages 632–647, 2007.

[99] Anduo Wang, Carolyn Talcott, Alexander J. T. Gurney, Boon Thau Loo, and Andre Scedrov. Reduction-based formal analysis of bgp instances. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 283–298, 2012.

[100] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Formal semantics and automated verification for the border gateway protocol. In *NetPL*, March 2016.

[101] J Whaley. Javabdd. `http://javabdd.sourceforge.net/index.html`.