Verification and Synthesis for Intent-based Networking

By

Sankararam Kausik Subramanian

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

Doctor of Philosophy
(Computer Sciences)

at the
University of Wisconsin-Madison
2020

Date of final oral examination: 07/15/2020

This dissertation is approved by the following members of the Final Oral Committee:

Srinivasa A Akella, Professor, Computer Sciences
Loris D'Antoni, Assistant Professor, Computer Sciences
Justin Hsu, Assistant Professor, Computer Sciences
Parmesh Ramanathan, Professor, Electrical and Computer Engineering

I would like to dedicate this thesis to Raman tata, who incited curiosity in my mind.

Acknowledgements

This thesis is the result of serendipitous events. My flatmate decided to take the program verification and synthesis course, and I just tagged along with him so that I would have company in the course. Genesis was my course project and led to a great collaboration between my advisors and eventually, this thesis.

I have immense gratitude for my advisors Aditya Akella and Loris D'Antoni. I was interested in networking from the start, and Aditya gave me an interesting problem to start my research, which jumpstarted my PhD. I am always amazed by his "big-picture" thinking, his dedication to research, and very grateful for his encouragement over the years. I could always knock on his door and discuss if a certain problem is worth tackling, and how we could bring research from other fields to help solve the problem. Loris saw my potential in his course, and I admired his attention to precision—formulating a networking problem and emphasizing on the correctness of our approach. I was also impressed by how he picked up networking along the way! Working in an inter-disciplinary field is hard, and my advisors complimented each other perfectly and kept me motivated throughout the journey.

I would also like to thank my undergraduate advisors from IIT Bombay - Puru Kulkarni, Umesh Bellur and Kameswari Chebrolu for giving me a glimpse of how research works and inspiring me to pursue a PhD in networking.

I did various internships over my PhD which gave me different perspectives about research in the industry and taught me skills I didn't pick at school. I want to thank JK Lee, Robert Soulé and Changhoon Kim at Barefoot Networks for giving me the first taste of working on cutting-edge technologies (P4), the first-hand experience helped in my research. I would also like to thank Andrey Rybalchenko and Nuno Lopes at Microsoft Research Cambridge, it gave me a sense of how research can be used to drive innovation in the real world. Finally, I would like to thank Hyojeong Kim, Mahesh Maddikayala and James Zeng at Facebook. I spent 3 months as an intern, learning how to write code for production networks, and then as a research collaborator. Facebook gave me a first-hand experience on the difficulties faced while designing and maintaining massive networks, and gave me a better appreciation of the challenges. These internships also taught me the art of teamwork, working with people to realize certain goals.

I would also like to thank the professors at Wisconsin who interacted with regularly, Justin Hsu for agreeing to be on my thesis committee and giving me feedback on Zeppelin in its early stages and my thesis, Aws Albarghouthi for giving me opportunities to review networking + PL papers, Paul Barford for guiding me through the teaching assistantship in my first year. I would also like to thank Nick Balster, whose course on teaching was one of the most delightful experiences of my academic life. The course sparked an interest for teaching and mentorship in me (I gave a small networking lecture as John Lennon in the course!). I would also like to thank Parmesh Ramanathan for serving on my thesis committee as well.

On the personal side, I would like to thank my parents and my sister for their unwavering support during the ups and downs of my PhD. My parents instilled in me the drive for academic success and accepted my life choices along the way. I would like to thank my aunt Vidya and uncle Mahesh would helped me when I first arrived to Madison. I would like to thank everyone in the WISR group who guided me and gave me company while getting coffee from the sixth floor and listened to my hare-brained ideas—Aaron Gember Jacobson and Raajay Viswanathan, who guided me initially on my research and helped me get it off the ground, Junaid Khalid, who showed me the art of enjoying life along the PhD, the Tycoons Archie Nidhi, Arjun Singhvi and Kshiteej Mahajan who were always open to discussing any topic, ranging from our research to if RCB will ever win the IPL (answer: probably not), Varun Chandrasekaran and Ayon Sen for the company to the national parks, Shaleen Deep, Arjun Bala and Adarsh Kumar for badminton, Atharva Kelkar for the bike rides, and, Anshul Purohit, Nivetha Vadivelu and Venkatesh Srinivasan for the delightful company in my first two years of my PhD and then in California. I would also to thank my California/IITB/London friends - Deepali Adlakha, Divyam Bansal, Guna Prasaad, Nilesh Kulkarni, Pranali Yawalkar, Rohan Das, Vipul Venkataraman and Vishaal Mohan for giving me company on my travels (meaning driving me around), or to the nearest Indian restaurant for eating chaat, or to listening to my current passion which keeps changing over time. It is virtually impossible to write every name who helped me along the way.

My final thanks is to my constant companion - rock music. Rock music kept me company over the years, most notably Pink Floyd, Genesis, Led Zeppelin, The Beatles, Porcupine Tree, Jethro Tull, Phish etc. It is a testament to networking that I have over 5000 songs from decades ago on my wrist, ready to be played.

Abstract

Management of modern data center, campus, and wide area networks have become increasingly complex. These networks must satisfy complex security, availability, and performance requirements. Moreover, these networks deploy a plethora of networking technologies, ranging from routers running distributed routing protocols like OSPF and BGP to programmable switches which can be programmed from a central controller using a standardized API. Consequently, configuring or programming networks to satisfy complex requirements has become a daunting task. This thesis focuses on the vision of *intent-based networking*: operators must be able to specify *what* the network must do rather than *how* these requirements are realized. In this thesis, we present three works: (1) QARC: a new network abstraction which can be used to verify network load violations in distributed control planes, (2) Genesis: a general extensible framework which synthesizes SDN forwarding tables using SMT solvers, and (3) Zeppelin: A two-phase synthesis approach for generating policy-compliant distributed router configurations.

Table of contents

Abstract **iv**

List of figures **ix**

List of tables **xii**

1 Introduction **1**
1.1 Intent-based Networking. 1
1.2 Our Contributions 3
1.2.1 QARC: Detecting Network Load Violations for Distributed Control Planes 3
1.2.2 Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks 3
1.2.3 Zeppelin: Synthesis of Fault-Tolerant Distributed Router Configurations 4

2 QARC: Detecting Network Load Violations for Distributed Control Planes **5**
2.1 Motivation 7
2.1.1 Network Control Planes 8
2.1.2 Control Plane Abstractions 9
2.1.3 ARC vs. Minesweeper 9
2.2 QARC (ARC with Quantities) 11
2.2.1 Problem Definition 12
2.3 QARC Encoding 13
2.3.1 Flow Constraints 14
2.3.2 Distance Constraints 15
2.3.3 Load Balancing Constraints 16
2.3.4 Failure Constraints 19
2.4 Verification using QARC 20
2.5 Optimizations 21

2.5.1 ETG Minimization . . . 21
2.5.2 Parallel Verification . . . 22
2.6 Upgrading Link Capacities in QARC . . . 23
2.7 Limitations . . . 24
2.8 Evaluation . . . 25
2.8.1 Verifying Real Networks . . . 26
2.8.2 Verification Performance . . . 28
2.8.3 QARC Optimizations . . . 29
2.8.4 Upgrade Performance . . . 30
2.9 Other Related Work . . . 31

3 Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks 33
3.1 Preliminaries and Policies Supported . . . 35
3.2 Data Plane Synthesis . . . 38
3.3 Synthesis of Tenant Policies . . . 40
3.3.1 Network Forwarding Model . . . 40
3.3.2 Reachability . . . 42
3.3.3 Waypoint Traversal . . . 43
3.3.4 Isolation . . . 43
3.4 Synthesis of Operator Policies . . . 44
3.4.1 Link and Switch Table Capacity . . . 44
3.4.2 Traffic Engineering . . . 45
3.5 Handling Failures Gracefully . . . 46
3.5.1 Dataplane Resiliency . . . 46
3.5.2 Minimal Repair . . . 49
3.6 Tactics . . . 50
3.6.1 Restricted Tactic Syntax . . . 51
3.6.2 Modified Synthesis Algorithm with Tactics . . . 52
3.7 Divide-and-Conquer Synthesis . . . 54
3.8 Evaluation . . . 57
3.8.1 Baseline Synthesis Performance for Tenant Policies . . . 58
3.8.2 Baseline Synthesis Performance for Operator Policies . . . 59
3.8.3 Tactic Reductions . . . 60
3.8.4 Divide-and-Conquer (DC) Synthesis Performance . . . 61
3.9 Related & Future Work . . . 62

4 Zeppelin: Synthesis of Fault-Tolerant Distributed Router Configurations 65
4.1 Overview . . . 68
4.2 Problem Definition . . . 70
4.2.1 Routing Model . . . 70
4.2.2 Policy Support . . . 72
4.2.3 Synthesis of policy-compliant configurations . . . 74
4.3 From Policies to Connectivity-Resilient Configurations . . . 75
4.3.1 A two-phase approach . . . 76
4.3.2 Synthesizing Path-compliant Intra-domain Configurations . . . 77
4.3.3 Synthesizing Path-compliant Inter-domain Configurations . . . 80
4.4 From Waypoint Policies to Policy-Resilient Configurations . . . 81
4.4.1 Problem Setup . . . 81
4.4.2 Intra-domain Policy-Resilient Waypoint Compliance . . . 82
4.5 Increasing Resilience through Domain Assignment . . . 87
4.5.1 Searching Assignments with MCMC . . . 88
4.5.2 The Cost of a Domain Assignment . . . 89
4.6 Evaluation . . . 89
4.6.1 Single Domain End-to-end Performance . . . 90
4.6.2 Resilience of Intra-domain Configurations . . . 91
4.6.3 Dynamic Domain Assignment Performance . . . 93
4.6.4 Comparison to SyNET . . . 93
4.7 Related Work . . . 94

5 Conclusion and Future Work 97
5.1 QARC . . . 97
5.2 Genesis . . . 97
5.3 Zeppelin . . . 98

References 99

Appendix A QARC Proofs 108
A.1 Verification Proofs . . . 108
A.2 Minimization Proofs . . . 110
A.3 Upgrade Proofs . . . 111

Appendix B Genesis Proofs 113
B.1 NP-completeness Proof of Enforcing Isolation Policies . . . 113

B.2 Resilience Transformation Proofs 114
B.3 Tactics Proofs 115

Appendix C Zeppelin Proofs **117**
C.1 Intra-domain Configuration Synthesis Proofs 117
C.2 Policy-Resilient Configuration Synthesis Proofs 121
C.3 Domain Assignment Proofs 125

List of figures

1.1 Intent-based Networking Vision 2

2.1 (a) Example control plane with OSPF and BGP. The boxes denote the routing processes, the numbers on links are OSPF weights, and the redistribution cost of OSPF to BGP is 1. There is an ACL installed at router B for traffic class $S-T$. (b) ARC ETGs for (a). 8
2.2 Network load verification times for ARC and Minesweeper for networks N1 and N2 and varying number of traffic classes. 10
2.3 ECMP behavior of node $C.1_O$ under two different scenarios for the partial $R-T$ ETG from Figure 2.1(b). Note that Sum of Flow in the network weighted with edge weights is the distance of the path taken by the flow for both scenarios. 17
2.4 Example OSPF network where a link gets overloaded under 1 and 2 link failure scenarios. The capacities in green are the minimum link capacity additions computed by QARC to ensure no violations occur under ≤ 2 failures. 23
2.5 Variation % for different ISP and datacenter networks for 1 and 2 link failure scenarios. 27
2.6 Verification time (log scale) sorted by network links for different WAN and datacenter networks for 1 and 2 link failure scenarios. 28
2.7 Capacity upgrade computation time 31

3.1 Genesis in a multi-tenant datacenter setting of a network containing several VMs and middleboxes. The network operator translates the tenant specifications and network resource policies to Genesis policies and Genesis synthesizes switch forwarding rules which are installed by the SDN controller. 36
3.2 Values of the *Fwd* and *Reach* relations of the network forwarding model for the policies specified in the figure. The blue and red arrows indicate the paths of packet classes 0 and 1 respectively according to the model. 41

3.3 (a) Resilience Transformation for $pc_1 || pc_2$ for providing 1-resilience. The dotted lines represent traffic isolation policies, while the solid lines represent link isolation. (b) Example of a sufficient transformation for 1-resilience in the case of a link-isolation policy. 49
3.4 Fat Tree Topology with three-level hierarchy. 50
3.5 Total synthesis time (log scale) for isolation workloads over range of packet classes and different tenant-group sizes. 57
3.6 Average synthesis time per packet class versus topology size for isolation workloads with the ratio of packet classes to number of edge-aggregate links 0.25 and 10 low bandwidth links in the topology have capacity policies. . . 61
3.7 Average synthesis time per packet class versus topology size for isolation workloads w/o different tactics with the ratio of packet classes to number of edge-aggregate links 0.25. 62
3.8 CDF for speedup achieved by divide-and-conquer synthesis. 62

4.1 Two-phase process for generating a control plane with failure-tolerance properties 66
4.2 Example demonstrating the two-phase synthesis approach. Genesis produces policy-compliant paths (red and blue), which are used as input by Zeppelin to synthesize the OSPF and BGP configurations. 69
4.3 Different configuration parameters in Zeppelin 70
4.4 Zeppelin Policy Support 73
4.5 Example illustrating the use of static routes. 78
4.6 Examples of BGP and static route configurations for inter-domain routing given domain assignment Θ. 80
4.7 Example of waypoint-compliant edge weights and example of a routing loop caused by a static route. 83
4.8 Example non-resilient configuration for the red paths provided by Genesis for $\mathbb{W} = \{r_1, r_2\}$. If link $r_1 \rightarrow t$ fails, a routing loop is formed at $r_0 \rightarrow r_1 \rightarrow r_3 \rightarrow r_0$, no traffic reaches t. The configuration is 1-resilient waypoint-compliant when the $s \rightarrow r_2$ weight is set to 1. 86
4.9 End-to-end synthesis time for waypoint policy workloads for varying number of paths and different number of waypoint sets. 91
4.10 Connectivity-resilience: PC vs baseline. 91
4.11 Policy-resilience scores of 2-WC, 1-WC, and PC synthesis for varying waypoint workloads. 92
4.12 MCMC Evaluation for varying number of paths. 92

4.13 Comparison of synthesis times of Zeppelin and SyNET for OSPF + Static route workloads. 93

B.1 The switch topology T. All circles represent switches and all reachability policies are s to d 113

C.1 Construction for reduction to Vertex Cover. 118
C.2 Construction for reduction to Graph Coloring. 125

List of tables

2.1 QARC variables 14
2.2 QARC constants 14
2.3 Networks which experience link overload for one or more TMs (>0%) and more than 10 of 20 TMs (>50%) at $k = 1$ and $k = 2$. 26
2.4 Percentage overload violations for 1-link failures for datacenter networks with Fbflow and Gravity traffic matrices 26
2.5 Verification times (s) for $k = [1,4]$ link failures. 29
2.6 Speedups due to optimizations for $k = 1$ failures. 30

3.1 Genesis Policy Support with Genesis Policy Language (GPL) syntax 37
3.2 Average synthesis time per class for waypoint policies with increasing number of waypoints. 60
3.3 Synthesis times for workloads on a 80-node fat-tree topology with different optimization objectives. 61

Chapter 1

Introduction

Modern networks, including data center, campus, and wide area networks, must satisfy increasingly complex security, availability, and performance requirements to meet the needs of a diverse group of users and applications. Additionally, these requirements could conflict with each other: higher performance comes at the cost of security. On the other hand, there is a plethora of networking technologies deployed in the real world. Many networks continue to rely on traditional routing protocols (e.g., OSPF, BGP, and MPLS), which require distributed configurations consisting of thousands of lines of vendor-specific syntax with hundreds of dependencies. With software-defined networks (SDNs), routing control is centralized, but programmers must still specify the precise actions (forward, drop, flood, etc.) to apply to individual traffic classes in order to realize complex policies.

Consequently, configuring or programming networks to satisfy complex requirements has become a daunting task. In fact, we see major network outages caused by errors incurred while programming the network, which can lead to significant loss of revenue. Operators also need to deal with device (link/switch) failures on a regular basis, and at the same time, provide Service-Level Agreements (SLAs) to tenants. Thus, operators desire to be able to check and enforce requirements under different network conditions. Failures further exacerbates the task of network management.

1.1 Intent-based Networking.

In an ideal scenario, the operator must be able to configure the network using high-level specifications, maybe even in natural language. For instance, a functional requirement may be to “allow communication between subnet A and subnet B”. To satisfy this requirement, the operator would need to write 10s of lines of code/configuration across the network devices. To simplify the task of satisfying functional requirements and reduce the likelihood of errors,

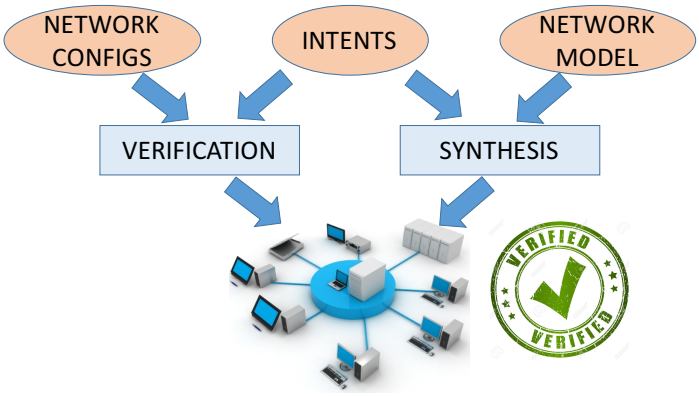


Fig. 1.1 Intent-based Networking Vision

we want to eliminate the need for operators to program the network at a low level. Operators must be able to specify *what* the network must do rather than *how* these requirements are realized; this is known as *intent-based networking* [3]. By raising the level of abstraction, intent-based networking enables more effective and streamlined management of networks. This removes the human element from the loop and network operators can achieve complete management automation. Our vision of intent-based networking is illustrated in Figure 1.1.

Verifying Intents. One aspect of intent-based networking is checking whether the current network satisfies the operator-specified intents at all times. While our vision tries to remove the human from the loop, humans inadvertently can affect the network in some form. For instance, if certain intents are not being satisfied, traditionally, the operator would manually fix the configuration, which could introduce new bugs. Thus, we must be able to verify existing networks efficiently to be able to detect errors under different network conditions. Verification is also useful to detect bugs in implementations of intent-based frameworks where human error can creep in. While there has been work on network verification in recent times, they are mainly focused on qualitative intents, i.e., intents which have a boolean answer. Verifying quantitative intents is important, i.e., intents which depend on quantities like bandwidth and latency. Quantitative intents play a significant role in multi-tenant datacenters where the network must satisfy tenant SLAs on performance and availability. Furthermore, the operator must also be able to reason about SLAs under different failure scenarios.

Synthesizing from Intents. Ideally, we want to raise the level of abstraction at which a network operator deals in while managing the network. Raising the level of abstraction however shifts the burden of implementation from the operator to the network management system; from a set of intents, the system must automatically synthesize an implementation which can be deployed to the network (SDN, legacy, etc.). There are numerous factors to consider in the design of such a system. Most importantly, the system must be *provably*

correct to eliminate the room for configuration errors, even under failure scenarios. Another important consideration is *generality*, if there exists a "one-size-fits-all" solution to network management to handle the diverse intents seen in different networks. Sythesis must also be *efficient*, which can be an important criteria for deployability.

1.2 Our Contributions

In this thesis, we present novel verification and synthesis frameworks to realize the vision of intent-based networking.

1.2.1 QARC: Detecting Network Load Violations for Distributed Control Planes [85]

In Chapter 2, we focus on verification of quantitative properties for distributed network control planes. Specifically, we dealt with the detection of network load violation: given a set of traffic demands, can any of the network links become overloaded (utilization exceeds capacity) which could lead to packet losses and increased latencies? We also want to proactively detect these load violations for different failure scenarios that can occur in the real world. Our abstraction QARC extends a state-of-art abstract representation of the network control plane (ARC) to model traffic load on links under any failure scenario. We then formulate a Mixed-Integer Linear Program (MILP) to check if network links are overloaded (utilization exceeds capacity) under different failure scenarios or certify if the network is free of load violations. QARC can model different routing protocols like OSPF and BGP and distributed load balancing strategies like ECMP/WCMP. We also devise various seamless extensions atop QARC to improve verification efficiency and compute minimal network capacity upgrades. We show that QARC can find violations in real-world networks in under an hour.

1.2.2 Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks [86]

In Chapter 3, we introduce Genesis, a general and extensible network management system for multi-tenant Software-defined (SDN) datacenter networks. Genesis allows rich policies to be specified declaratively and eliminates the need for operators to program the SDN data plane. Prior work for programming SDNs were too low-level, not expressive enough to specify diverse intents (e.g., support for hyperproperties: multiple traffic classes dependent

on each other), and were not easily extensible to new intents. We address all these challenges by abstracting the network forwarding behavior and leveraging formal reasoning foundations of constraint solving together with fast SMT solvers to synthesize provably correct switch forwarding tables. Our use of SMT solvers was further motivated by the theoretical complexity of enforcing intents (many intents were NP-complete, for e.g., isolated paths). To make synthesis more tractable, Genesis incorporates a novel search strategy using regular expressions to specify properties which use the structure of datacenter networks, and a divide-and-conquer synthesis procedure which exploits the structure of policy relationships. Using Genesis and it's network-specific optimizations, we are able to synthesize forwarding tables for medium sized datacenter networks comprising of 80 routers.

1.2.3 Zeppelin: Synthesis of Fault-Tolerant Distributed Router Configurations [87]

One of the limitations of Genesis was that it was constrained to SDNs, while many networks run "legacy" distributed routing protocols like OSPF and BGP because these protocols are scalable and robust to failures. In Chapter 4, we present Zeppelin, a tool which can provide support for the intents supported by Genesis to synthesize distributed OSPF and BGP control planes. Since failure handling is distributed in these networks, we augment our synthesis algorithms to provide compliance for intents under small link failures. Zeppelin uses a novel two-phase synthesis approach: (1) it generates paths from intents using Genesis, and then (2) generates intra-domain (shortest-path OSPF) and inter-domain (BGP) router configurations that induce the forwarding state synthesized by Genesis and provide high resilience. For the second phase, we use fast linear programming (LP) solvers and stochastic search to perform better than state-of-art tools which attempt to synthesize control planes directly from intent. Using Zeppelin, we can synthesize highly-resilient and policy-compliant configurations for medium-sized topologies (with 80 routers) 2-3 orders of magnitude faster than state-of-art approaches which synthesize configurations directly from intent.

Chapter 2

QARC: Detecting Network Load Violations for Distributed Control Planes

Managing modern data center and ISP networks is an incredibly difficult task. Many of these networks serve a diverse set of customers who demand stringent guarantees from the network, such as high path availability and a variety of path-based properties (e.g., service chaining, and path isolation), and these requirements evolve over time. Thus, operators must face the challenge of programming a network that meets various requirements.

Various frameworks have been developed to ease the operators' task of programming the network to ensure requirements are met. Intent-based programming [3], where operators specify what they want the network to do instead of worrying about how the network must be configured, has made synthesizing network control and planes simple [29, 86, 11, 12, 87]. Using these tools, operators need to specify the network requirements at a high-level, and compliant low-level network implementations are synthesized automatically. Other frameworks validate if the current network satisfies important properties [7, 42, 32, 37, 31, 10], and automatically take corrective action otherwise [30, 36].

While these tools are invaluable, they focus on *qualitative* properties—informally, path properties that are all variants of reachability. In contrast, *quantitative* properties, which are also central to network management, have received little systematic treatment. One such property, which we focus on, pertains to meeting demand, i.e., *can a given network (topology and control plane) accommodate input traffic demand without any link becoming overloaded?*

The focus on qualitative properties is well justified, because violation of such properties can lead to serious policy violations, e.g., broken isolation, circumventing firewalls, or network partitions. But violation of quantitative properties can have equally serious implications. For instance, when network links become overloaded, customers' applications may suffer

from increased losses and latencies, violating customer guarantees and impacting operator revenues.

We present an intent-based framework that helps programmers verify whether a network meets a complex quantitative property—absence of link overload when network links become unavailable due to failures. We believe that our *verification* framework forms the first step in the much harder challenge of developing an intent-based framework for generating compliant network control planes for qualitative *and* quantitative properties.

Verification. Verifying link overload is non-trivial. First, link failures are common in networks [43] due to various factors, e.g., human error and device overheating. Thus, even if link loads are low now, significant overload may occur when one or more links fail. In datacenter networks, packet losses due to high load (i.e., congestion) are common [97]. Ideally, we must *proactively verify for potential overload*: does there exist a failure where some link is overloaded? This is difficult because number of failure scenarios increase exponentially with number of simultaneous link failures.

Second, checking for overload needs an estimated model of traffic volumes, but actual traffic volumes may vary substantially around such estimated values [76]. The network's control plane may not be programmed to accommodate such variations, leading to overload when traffic spikes unexpectedly along certain paths. Thus, verification must account for *variations* in measured traffic volumes. Unlike the set of failures which can be enumerated, the set of traffic demands to consider could be infinite (as traffic quantities are rational).

Third, the control planes in most networks are *distributed* in nature and use a mix of different routing protocols [40, 16], such as Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP), configured in low-level languages. The complexity of the control plane designs [40, 13, 16] and the interactions among the constituent routing protocols make it difficult to reason about the paths induced in a network under failures, and hence, about potential link overload.

Our first contribution is a verification framework that, given a network, its distributed router configurations, and an input traffic matrix with variable traffic volumes, identifies failure scenarios that can cause overload on some network link. Our framework also allows operators to focus the verification on failure events that occur with a certain minimum likelihood or involve k or fewer links.

To conduct verification, we need a model of how the distributed control plane reacts to failures and recomputes paths, and how traffic load is spread across recomputed paths. Furthermore, analyzing the model to determine potential load violations should be fast to enable quick corrective action. To this end, we develop QARC for quantitative analysis of control planes, with a focus on analyzing link overload. QARC builds on a weighted

digraph-based control plane abstraction, and couples it with flow quantities and a novel mixed integer linear program encoding. QARC computes if links can become overloaded under some failure without enumerating failures or traffic matrices that adhere to a given bounded variation; without materializing paths under each failure; and without having to analyze all links and considering the load contributed by all source-destination pairs. These optimizations to verification are our second contribution, which help achieve substantial speedup of $5-800\times$ over SMT-based state-of-art verifiers [10].

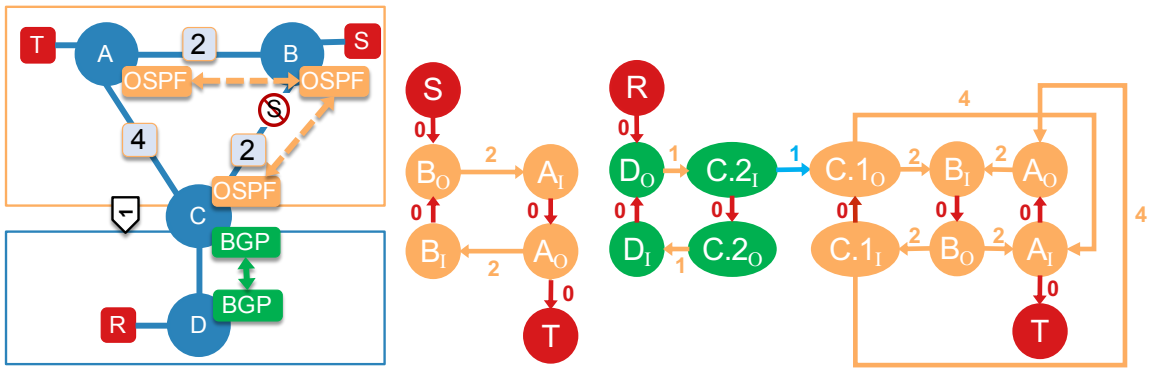
Upgrade. Once verification has identified a potential load violation, it is crucial to understand how to avoid overload to mitigate impact on production traffic as much as possible. There are many aspects of a network's design and operation which can cause load violations, e.g., network topology, configuration, input traffic matrices, etc. It is cumbersome for operators to manually identify the root cause of a violation and reconfigure the network to ensure load violations do not happen in practice. Instead, we propose an intent-based approach where operators specify the input traffic characteristics and set of failure scenarios, and QARC will upgrade the link capacities such that no load violations will occur under these scenarios. Moreover, QARC will compute the *minimal* capacity upgrade to prevent unnecessary overprovisioning.

Analysis of real networks. Today, very little is known about how robust real-world network designs and control plane configurations are in avoiding overload under observed and expected traffic patterns. We conduct a detailed study of potential link overload in production networks.

We apply QARC to 112 data center networks of a large service provider, and 86 ISP networks from the Topology zoo. We use a mix of real and synthetic control plane configurations. We apply a variety of realistic traffic matrices and traffic volume variations. We find that *70%* of the networks experience link overload under 1 or 2 link failures with different traffic characteristics and variation. We also find that the ISP networks in our study are susceptible to link overload under failures if overall traffic increases by 2-12%, while datacenter networks require 5-35% to experience link overload. We also show QARC is practical: QARC can verify real-world networks we study in under an hour (§2.8).

2.1 Motivation

We provide an overview of distributed network control planes, control plane abstractions and their shortcomings for reasoning about network load violations.



(a) (b)

Fig. 2.1 (a) Example control plane with OSPF and BGP. The boxes denote the routing processes, the numbers on links are OSPF weights, and the redistribution cost of OSPF to BGP is 1. There is an ACL installed at router B for traffic class $S-T$. (b) ARC ETGs for (a).

2.1.1 Network Control Planes

A distributed network has two components: the control and the data plane. The control plane exchanges information with routers and decides the best path for every packet, these best paths are installed in the data plane. In a distributed control plane, each router runs one or more routing processes, each process implementing a path selection algorithm, defined by a standard routing protocol—e.g., OSPF [65], BGP [74], and RIP [58]. A routing process will receive a set of routing advertisements from different routing processes: could be from the same router or neighbouring routers, and could be of the same or different routing protocol. Each routing process's path computations are based on different protocol-specific configuration parameters and advertisements. Each routing process decides the best path(s) and could advertise the best path(s) to other routing processes. The steady-state outcome of the control plane's computation is a set of forwarding rules computed at each router, which encode the next-hop for different traffic classes, i.e., source-destination subnets. Operators typically decide which protocols and configuration to use based on network design and operational policies they wish to enforce [59] like reachability, waypointing etc.

Figure 2.1(a) shows a toy distributed control plane spanning four routers. Two routers run OSPF, one runs BGP, and one runs both and is configured to redistribute routes from OSPF to BGP. The OSPF edge weights are configured to support load balancing for $R-T$ traffic, while an Access Control List (ACL) is configured at router B to block $S-T$ traffic for a security policy. Distributed control planes are programmed at a low-level by configuring routers using vendor-specific languages (e.g., Cisco IOS [1]), either manually or using automated tools [88, 11, 29, 87].

Another mechanism commonly used in distributed control planes is load-balancing traffic between endpoints. ECMP (Equal Cost Multi-Path routing) [50], or its weighted variant

(WCMP [95]), are used to distribute traffic across multiple paths between endpoints in the network. In an ECMP router, traffic is split *equally* among the "best paths", where the best paths are chosen based on routing metrics.

2.1.2 Control Plane Abstractions

Analyzing whether a network satisfies certain properties over paths, qualitative or quantitative, requires knowing how the forwarding paths of the network. Understandably, inferring this based on low-level router configurations is difficult as the forwarding paths is the result of complex distributed computations. Thus, many frameworks [32, 37, 7, 10, 92, 31] have been developed to model the distributed network control logic, so that different inferences can be made about network forwarding—under failures—without actually deploying the network configurations. These models are used widely in verifying qualitative properties, e.g., middlebox traversals.

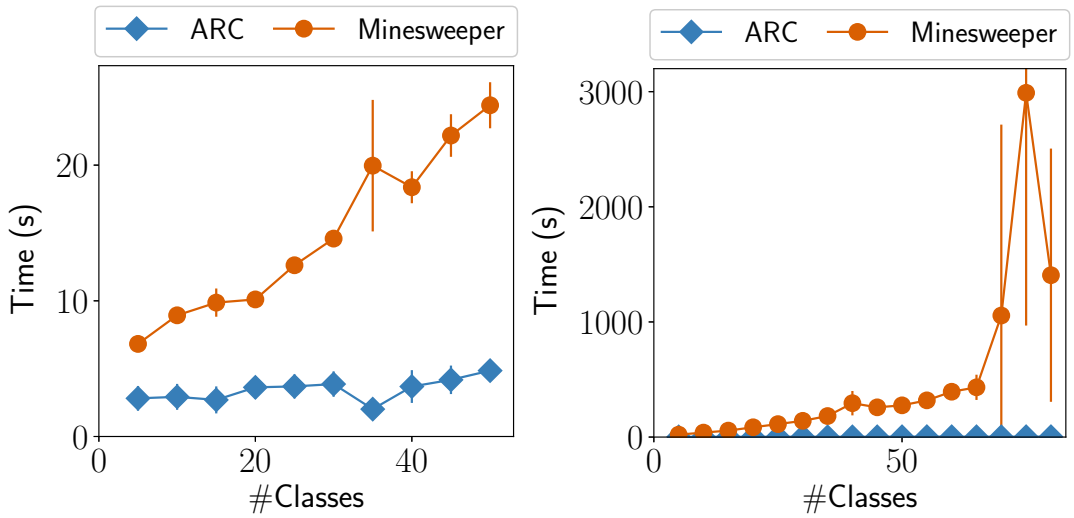
However, verification of network overload, a quantitative property, is challenging in aspects that the state-of-art frameworks cannot handle. Firstly, verifying this property requires a model that can reason about how much traffic a traffic class sends on a path (or multiple paths for ECMP/WCMP). Moreover, the model needs to reason about the collective load imposed by *all* traffic classes in the network. Contrarily, verifying qualitative properties only requires reasoning about small subsets of traffic classes.

Secondly, operators want to understand if the network can handle load under different failure scenarios. While in the worst-case scenario where all devices fail together the network will be overloaded, the probability of such scenario occurring is very small. Thus, the operator may constrain the set of requisite failure scenarios—e.g., only 1 and 2-link failures. The number of failure scenarios grows exponentially with number of failed links. Thus, the approach adopted by some control plane models, e.g., Batfish [32], of enumerating all failure scenarios can be *prohibitively* expensive. Therefore, we require our network model to symbolically encode network routing state even under failures.

2.1.3 ARC vs. Minesweeper

Two networks models are amenable for verifying network overload under failures: ARC [37] and Minesweeper [10]. Our work builds on ARC and in this section we motivate this choice.

The ARC network model uses the fact that most routing protocols use a *cost-based path selection algorithm*. E.g., OSPF uses Dijkstra's algorithm to compute minimum cost paths from a source to all destinations, where each link has a cost; and BGP selects paths based on numeric metrics, most importantly, the AS path length. ARC abstracts the different routing



(a) N1 (b) N2

Fig. 2.2 Network load verification times for ARC and Minesweeper for networks N1 and N2 and varying number of traffic classes.

protocols, static routes, and ACLs based on the above principle, and models the network's collective forwarding behavior under failures using a set of weighted directed graphs (called ETGs) satisfying the *path-equivalence* property: under *any failure scenario*, the path taken by the traffic will correspond to the *shortest weighted path* in the graph. Figure 2.1(b) illustrates the ETGs for the two traffic classes (R-T and S-T) for the network shown in Figure 2.1. ARC cannot model every set of router configurations as certain routing constructs like BGP local preferences do not adhere to the cost-based path selection principle.

Minesweeper [10] encodes the converged routing state of the control plan using SMT (Satisfiability Modulo Theories) formulas [27] and supports iBGP and BGP local preferences, which ARC cannot handle. Minesweeper is primarily used to verify qualitative properties dealing with a single or small subset of traffic classes under 1-link or no failure.

Let us consider ARC v/s Minesweeper for detecting link overload under no failures. Using ARC, we can compute the exact network paths taken by the traffic classes using the polynomial-time Dijkstra's algorithm and thus compute individual link utilizations. On the other hand, for Minesweeper, due to the symbolic nature of the network abstraction, we need to use the SMT solver to find the network paths and compute utilizations. Handling network failures further slows down verification. We compare verification performance by using the Minesweeper and ARC abstractions for $k = 1$ link failure scenarios in Figure 2.2. We can observe that ARC is significantly faster than Minesweeper, achieving $5\times$ speedup for N1 and $20 - 800\times$ for the larger N2 network.

Moreover, the ARC abstraction offers certain key benefits for network transformations [69] which can help reduce the search space efficiently and enables even faster quantitative verification than Minesweeper (§2.5). For example, in ARC, we know a certain path will not be traversed if a shorter path exists in the network (by simply comparing the path cost). This property can be used to reduce the search space—both by limiting which links to consider as candidates for overload, and by ignoring the contributions of some source-destination pairs' traffic volumes—in scenarios when a small number of links can fail together. In Minesweeper, the paths are symbolic (represented by SMT constraints), and thus, we cannot a priori tell which path would be preferred.

Since ARC is a more natural fit for quantitative analysis, ARC's natural attributes support effective speedup of verification, and ARC has near-universal control plane coverage, we adopt it as the basis for our framework. In § 2.2, we extend ARC to model (1) paths under failures without enumeration, (2) distribution of load on equal cost paths (due to ECMP or WCMP), and (3) variation of traffic volumes.

2.2 QARC (ARC with Quantities)

ARC. ARC abstracts the set of router configurations using a set of weighted directed graphs called *extended topology graphs* (ETGs). For each traffic class, ARC constructs one ETG to model the behavior of the network routing protocols and interactions among the routers for the traffic class (shown in Figure 2.1(b)). In the ETG, each vertex corresponds to a routing process; there is one incoming (I) and one outgoing (O) vertex for each process. Directed edges represent possible paths enabled by exchange of routing advertisements between connected processes. The weights of the edges of the ETG satisfy the *path-equivalence property*: under any failure scenario, the actual network path(s) are the shortest weighted path(s) in the ETG. Thus, the ARC abstraction can be used to compute the forwarding state of the network under any failure scenario, abstracting away the need to model route advertisements and path computations at each router. Note that the path-equivalence property of ARC can model ECMP-style load-balancing.

ARC was designed to use graph algorithms on the extended topology graphs (ETGs) to verify qualitative properties under *any* arbitrary failure scenario—e.g., a traffic class is connected under any k-link failure if the min-cut for the traffic class's ETG is $\geq k$.

QARC. QARC extends ARC and uses a mixed-integer linear program (MIP) encoding to add symbolic traffic quantities to the ARC ETGs to verify load properties under different failure scenarios.

Verification. Provisioning bandwidth is expensive, especially in wide area networks where adding new links is not an easy endeavor. Operators therefore want to keep link utilization high and, at the same time, tolerate link failures. QARC helps operators predict if the network will encounter load violations under failures. Operators have frameworks to periodically update routing configurations [88, 11, 87] and measure input traffic characteristics [55, 49, 44]. We envision QARC to be used for verification whenever traffic characteristics change significantly, or the control plane is reconfigured. Taking as input the new control plane and traffic matrix (between endpoints) with bounds on traffic variation, our verification detects if there exists a failure scenario leading to utilization exceeding capacity on any link.

Consider the network control plane in Figure 2.1(a) and the corresponding ARC ETGs in Figure 2.1(b). In this scenario, the operator sees a change in the input traffic: the current traffic for classes $R \to T$ and $S \to T$ are 80 Gbps and 30 Gbps, respectively. All links in the network have a capacity of 100 Gbps. When no links have failed, the traffic from $S \to T$ flows through the path $B \to A$, while the traffic from $R \to T$ is load-balanced by ECMP at C; 40Gbps traffic is sent through the two paths: $D \to C \to A$ and $D \to C \to B \to A$ (these are the shortest network paths in the ETGs). As we can observe, all links' utilizations are below capacity ($D \to C : 80$, $C \to B : 40$, $C \to A : 40$, $B \to A : 40 + 30 = 70$).

Suppose, the operator wants to inspect if the network, for the given traffic matrix, can experience some link becoming overloaded under a single link failure. Using our tool QARC, the operator can discover that when $C \to A$ fails, the traffic on $B \to A$ will be 110 Gbps, exceeding the link's capacity.

Going further, the operator can discover other single link failures by asking our verification tool to find 1-link failures *other than* $C \to A$. Likewise, the operator can discover sets of k-link failures that cause links to overload.

Upgrade. When verification detects a possible load violation, the operator can invoke our upgrade framework. Taking as input the network configurations and the traffic matrix, QARC computes a minimal set of links whose capacities need to be upgraded to avoid overload under failures. For the network in Figure 2.1, QARC suggests to increase the $B \to A$ capacity by 10Gbps to ensure no load violations occur under any 1-link failures.

2.2.1 Problem Definition

Let TC be a set of traffic classes, $N = (Routers, Links)$ be the network topology and $FL \subseteq Links$ be the failure scenario—i.e., the set of links that failed. For each traffic class $tc \in TC$, the routing processes compute paths from source to destination; the paths form a directed acyclic graph (DAG), which we call the *flow graph*. We represent tc's flow graph as $FG(tc, FL) = (V_{tc}, L_{tc})$. Traffic will not flow on failed links, thus $L_{tc} \cap FL = \emptyset$. ARC

constructs a weighted directed graph $ETG(tc)$ for each traffic class tc with the following property:

Theorem 2.2.1 (Path-equivalence [38]). *For every traffic class $tc \in TC$, if the destination is reachable from the source router in the actual network, then, after removing edges corresponding to failed links from $ETG(tc)$, the shortest path in the ETG from the source router to destination router is equivalent to the path computed by the actual network.*

Thus, the flow graph $FG(tc,FL)$ will be a sub-graph of $ETG(tc)$ and every path from source to destination in $FG(tc,FL)$ will be the shortest weighted path based on $ETG(tc)$ weights.

Since there are multiple paths in the flow graph, the traffic is split across different paths based on the load-balancing scheme deployed in the network. We define the flow function $F(l,tc,FL)$ as the amount of flow of tc on a link l in $FG(tc,FL)$. In ECMP, each router divides the total incoming flow equally among the outgoing links leading to shortest paths. For a node r, we define the set of nodes which have an incoming edge into r as $prev(r) = \{r' | (r',r) \in L_{tc}\}$, and the set of nodes connected by an outgoing edge from r as $next(r) = \{r' | (r,r') \in L_{tc}\}$. Thus, we can define ECMP behavior in terms of the flow function as:

$$\forall r. \forall r' \in next(r).\ F((r,r'),tc,FL) = \frac{\sum_{r'' \in prev(r)} F((r'',r),tc,FL)}{|next(r)|}$$

For each link $l \in Links$ and failure scenario FL, we define the utilization l under the failure scenario FL as $Util(l,FL) = \sum_{tc \in TC} F(l,tc,FL)$. We represent the link capacity of link l as $Cap(l)$. A network load violation occurs when a link's utilization exceeds its capacity.

Definition 1 (Verification). *Given a set of failure scenarios $\overline{FL}$, the verification problem is to find a failure scenario $FL \in \overline{FL}$ that leads to a network load violation—i.e., $\exists FL \in \overline{FL}.\ \exists l \in Links.\ Util(l,FL) \geq Cap(l)$.*

2.3 QARC Encoding

We now present the mixed-integer linear program (MIP) encoding the semantics of QARC. Our encoding makes new technical contributions. First, it extends ARC shortest-path-forwarding ETGs with flow quantities while accounting for failures and permitting bounded variation of traffic characteristics. This is crucial to determining link overload. Second, it *models flow being split among multiple shortest paths* even under failures, i.e., QARC

Table 2.1 QARC variables

Name	Description	Range
$Flow(e,tc)$	Fraction of traffic for class tc flowing on edge e	[0,1]
$\Delta(tc)$	Fraction of traffic variation for class tc	[0,1]
$\Delta(e,tc)$	Fraction of traffic variation for class tc flowing on edge e	[0,1]
$Dist(n,tc)$	Shortest path distance of node n to destination in ETG	$\mathbb{Q}$
$Fail(e)$	Link e ($\forall tc$) failure status ($1 \equiv$ failed)	$\{0,1\}$
$Load(e)$	Link e load status ($1 \equiv$ overloaded)	$\{0,1\}$

Table 2.2 QARC constants

Name	Description
TC	Set of all traffic classes
$\overline{FL}$	Set of all failure scenarios
$T(tc)$	Traffic sent by class tc ($T(tc) \in \mathbb{Q}$)
$W(e,tc)$	Weight of edge e in tc ETG
$Src(tc)$	Source node of traffic class tc
$Dst(tc)$	Destination node of traffic class tc
$In(n)$	Set of in-edges of node n in ETG
$Out(n)$	Set of out-edges of node n in ETG
$OEdges(tc)$	Set of all outgoing edges in tc ETG
$Links$	Set of all physical links in the network
$Cap(e)$	Capacity of link e ($Cap(e) \in \mathbb{Q}$)
Π	Threshold of total traffic variation

models ECMP, which is a key construct used in most networks. Third, it does not require disjunctions and only uses integer variables to represent failures and link overload, therefore enabling fast verification.

Table 2.1 and Table 2.2 describe QARC's variables and constants, respectively. For each section, the sentences within boxes state what property the presented constraints encode.

2.3.1 Flow Constraints

Traffic flows from source to destination along *active* links and flow is *conserved* at every node in the ETG.

The $Flow$ and Δ variables represent respectively the fraction of normal and variation of traffic flowing along the network links. For class tc and edge e, the actual traffic on the edge is $(Flow(e,tc)+\Delta(e,tc)) \times T(tc)$. The distinction of $Flow$ and Δ variables ensures the verification constraints (2.13) are linear while providing variation in traffic characteristics. For every ETG and corresponding traffic class tc, the outgoing flow at tc source and the

incoming flow at tc destination should equal $1+\Delta(tc)$ which are constrained separately:

$$\begin{aligned} \sum_{e\in Out(Src(tc))} Flow(e,tc) = 1 \qquad & \sum_{e\in In(Dst(tc))} Flow(e,tc) = 1 \\ \sum_{e\in Out(Src(tc))} \Delta(e,tc) = \Delta(tc) \qquad & \sum_{e\in\ In(Dst(tc))} \Delta(e,tc) = \Delta(tc) \end{aligned} \tag{2.1}$$

For all other nodes $n \notin \{Src(tc), Dst(tc)\}$ in the ETG, flow is conserved, i.e., incoming flow is equal to outgoing flow.

$$\begin{aligned} \sum_{e_{in}\in In(n)} Flow(e_{in},tc) \quad &= \sum_{e_{out}\in Out(n)} Flow(e_{out},tc) \\ \sum_{e_{in}\in In(n)} \Delta(e_{in},tc) \quad &= \sum_{e_{out}\in Out(n)} \Delta(e_{out},tc) \end{aligned} \tag{2.2}$$

Next, we need to accommodate failures. When a link fails, no traffic must flow on it:

$$Flow(e,tc) + Fail(e) \leq 1 \quad \bigwedge \quad \Delta(e,tc) + Fail(e) \leq 1 \tag{2.3}$$

Thus, if $Fail(e) = 1$, then $Flow(e,tc) = 0$ and $\Delta(e,tc) = 0$.

Operators can bound the individual variation for each traffic class and/or bound the total extra variation of traffic in the network by a threshold Π:

$$\sum_{tc\in TC} \Delta(tc) * T(tc) \leq \Pi \tag{2.4}$$

Given a solution to the above constraints, we construct the flow graph of tc by picking all edges e where $Flow(e,tc) > 0$, which indicate all the links traffic flows on.

2.3.2 Distance Constraints

A computed path in the real network matches the shortest path in the ETG between the corresponding source and destination (Theorem 2.2.1). Thus:

Traffic must only flow on shortest-distance ETG paths.

We formulate shortest path distances to the destination node of the ETG in an inductive fashion - the shortest path from a node must traverse through one of its neighbors, and thus, distance of node n can be defined inductively as the shortest among distances from the neighbors. Again, failures introduce complications, because distances can change under different link failures. To this end, we need to two sets of constraints. First, we use the following constraints to ensure that for every node n and traffic class tc, the value of the

variable $Dist(n,tc)$ is *at most* the distance of the shortest path that traverses only active links. For all $e = n \rightarrow n' \in Out(n)$:

$$Dist(n,tc) \leq W(e,tc) + Dist(n',tc) + \infty \times Fail(e) \tag{2.5}$$

The intuition is that when edge $n \rightarrow n'$ has failed and $Fail(n \rightarrow n') = 1$, the shortest distance from n will not depend on the path through n' as the equation will be satisfied trivially due to the large constant in front of $Fail(n \rightarrow n')$. The above constraints provide an upper-bound on *Dist* variables, but do not ensure that the variable values are exactly equal to actual shortest distances in the ETG. E.g., setting all *Dist* variables to 0 trivially satisfies Constraints (2.5).

Second, we use another set of constraints to ensure that the ETG traffic flow only uses the shortest paths in the network. (We will consider load-balancing in §2.3.3 and ignore it here). We illustrate the idea of our encoding using the example ETG in Figure 2.3. For the $Flow$ quantities, the cost of the path taken by the flow can be computed by the sum of $Flow$ on all ETG edges multiplied by the edge weights – i.e., $\sum_e W(e,tc) \times Flow(e,tc)$ (regardless of whether the flow is sent on the shortest path or not). The following constraint ensures that the flow is sent on the shortest distance path from source to destination of the ETG of tc:

$$Dist(Src(tc),tc) = \sum_{e \in OEdges(tc)} Flow(e,tc) * W(e,tc) \tag{2.6}$$

Since Constraint (2.5) guarantees upper bounds on the shortest distance, the above constraint will ensure the traffic will not be sent on a longer path. Also, by virtue of Constraints (2.5) and (2.6), the *Dist* variables for all nodes in the ETG will be exactly equal to the shortest distances to the destination. Notice, that *the constraint will guarantee this property even if traffic flows across multiple shortest paths*.

To ensure that the traffic variation only flows on shortest paths, we add the following constraint:

$$\forall e \in OEdges(tc).\ \Delta(e,tc) \leq Flow(e,tc) \tag{2.7}$$

By virtue of Constraint (2.6), $Flow(e,tc)$ is 0 on non shortest path links, and consequently, $\Delta(e,tc)$ will be 0.

2.3.3 Load Balancing Constraints

The outgoing $Flow$ and Δ of each node are *split* equally among the shortest neighbors connected by active links.

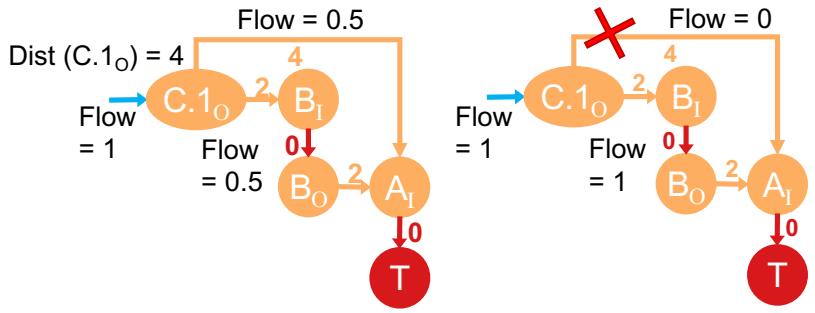


Fig. 2.3 ECMP behavior of node $C.1_O$ under two different scenarios for the partial $R-T$ ETG from Figure 2.1(b). Note that Sum of Flow in the network weighted with edge weights is the distance of the path taken by the flow for both scenarios.

Figure 2.3 demonstrates how ECMP operates under no failures, and the differences that arise when a link failure occurs.

At any router, by virtue of Constraints 2.1-2.7, traffic will never be sent along longer paths. The constraints presented in this section ensure traffic is split equally among the active links at a router which are on the currently available shortest paths to the destination. First, we show a simple disjunctive constraint that encodes the desired behavior and we then show how the same behavior can be captured without disjunction.

For node n and all outgoing edge pairs $e_1 = n \rightarrow n_1$ and $e_2 = n \rightarrow n_2$ in $Out(n)$, ECMP for $Flow$ (similarly Δ) is:

$$[W(e_1,tc) + Dist(n_1,tc) = W(e_2,tc) + Dist(n_2,tc)] \\ \wedge \neg Fail(e_1) \wedge \neg Fail(e_2) \implies Flow(e_1,tc) = Flow(e_2,tc)$$

The above constraints ensure that if the distance to the destination along two active outgoing edges is equal, then the flow on them will also be equal (if these edges do not lie on the shortest path, then flow will be 0 on both edges). However, these constraints require disjunctions and cannot be provided in this form to an ILP solver.

By introducing new rational variables, we can write constraints that express the ECMP load balancing constraints using only linear inequalities. This forms a *key innovation of* QARC *encoding*. Specifically, we define sets of variables $minFlow(n,tc) \in [0,1]$ and $maxFlow(n,tc) \in [0,1]$ to capture the minimum and maximum non-zero $Flow$, respectively, out of node n for traffic class tc. The non-zero flow restriction holds for flows along the shortest paths (flow along non-shortest paths will be 0), and thus we can impose constraints on $minFlow$ and $maxFlow$ to model ECMP routing. We also add similar constraints for the traffic variation Δ variables so that the total traffic adheres to ECMP.

Adding constraints for $maxFlow(n,tc)$ to be the maximum non-zero flow value is trivial, as the zero flow values will not affect the $maxFlow$ variable:

$$\begin{aligned} \forall e = n \to n' \in Out(n).\ Flow(e,tc) \leq maxFlow(n,tc) \\ \forall e = n \to n' \in Out(n).\ \Delta(e,tc) \leq max\Delta(n,tc) \end{aligned} \tag{2.8}$$

Adding constraints for $minFlow(n,tc)$ is trickier: to ensure that $minFlow(n,tc)$ is the minimum non-zero flow value, we need to know the flow values *for all possible failure scenarios*. To bypass this problem, we use the distance variables $Dist$ to identify the next-hops that lie on the shortest paths, and thus, have non-zero flows. Based on this insight, we impose the following constraints for all $e = n \to n' \in Out(n)$ to ensure that $minFlow$ variables have the correct values:

$$\begin{aligned} &\forall e = n \to n' \in Out(n).\ minFlow(n,tc) - Flow(e,tc) \leq \\ &\qquad \infty \times (W(e,tc) + Dist(n',tc) - Dist(n,tc) + Fail(e)) \\ &\forall e = n \to n' \in Out(n).\ min\Delta(n,tc) - \Delta(e,tc) \leq \\ &\qquad \infty \times (W(e,tc) + Dist(n',tc) - Dist(n,tc) + Fail(e)) \end{aligned} \tag{2.9}$$

First, notice that the quantities $W(e,tc)+Dist(n',tc)-Dist(n,tc)$ and $W(e,tc)+Dist(n',tc)-Dist(n,tc)+Fail(e)$ are always greater or equal than 0. Hence, there are three scenarios for each edge which are all encoded by the above constraint: (1) the edge e has failed, in which case $Fail(e) = 1$, the RHS of the constraint is infinity, and the constraint is trivially satisfied, (2) the edge e is not on the shortest path, thus $W(e,tc)+Dist(n',tc)$ is greater than $Dist(n,tc)$ and thus, the RHS is a large positive constant, and (3) the edge e is active and on the shortest path, so the RHS of the constraint is 0—i.e., $minFlow(n,tc) \leq Flow(e,tc)$. Therefore, $minFlow(n,tc)$ is smaller than all the non-zero edge flows (which can only flow on the shortest paths).

Thus, for a node n, we have two variables for the lower bound and upper bound of all the non-zero edge flows out of the node. For ECMP, we require all non-zero edge flows on the shortest paths out of a node to be equal, which can be enforced by ensuring the lower bound $minFlow(n,tc)$ and upper bound $maxFlow(n,tc)$ are equal (similarly for Δ):

$$\begin{aligned} minFlow(n,tc) = maxFlow(n,tc) \\ min\Delta(n,tc) = max\Delta(n,tc) \end{aligned} \tag{2.10}$$

Constraints 2.1-2.10 ensure the total flow to neighbors on the shortest paths are equal to one another, adhering to the ECMP load-balancing model. The above constraints can be modified by multiplying constant weight factors to $Flow$ variables to model Weighted Cost Multipathing (WCMP) [95].

2.3.4 Failure Constraints

Failure scenarios can be restricted by the operator, e.g., by number of links or likelihood of failures.

Operators may want to restrict the failure scenarios of interest ($\overline{FL}$) to make useful observations about how the network control plane reacts under failures. The following constraint restricts the number of link failures to k:

$$\sum_{e \in Links} Fail(e) \leq k \tag{2.11}$$

Theorem 2.3.1. *For every traffic class $tc \in TC$ and failure scenario $FL \in \overline{FL}$, Constraints 2.1-2.11 ensure that the flow Graph $FG(tc, FL)$ is a directed acyclic graph such that each path from $Src(tc)$ to $Dst(tc)$ is the shortest path in $ETG(tc)$, and for every router n in flow graph $FG(tc, FL)$, the flows on outgoing edges which lie on shortest paths from n to $Dst(tc)$ are equal., i.e., $\forall n_1, n_2 \in next(n).\ F((n, n_1), tc, FL) = F((n, n_2), tc, FL)$.*

Theorem 2.3.1 and Theorem 2.2.1 together prove that the flow graphs constructed by the QARC constraints faithfully represent the routing paths and flow distributions in the actual network.

Operators can also assign failure probabilities to individual links and restrict the search to scenarios that have probability above a threshold. Suppose, the operator has probabilities assigned to individual link failures ($P_{fail}(l)$) and is only interested in link failure scenarios that have joint probability above a threshold ω. The failure probabilities can be derived from telemetry data in real-world networks [61]. We assume that link failures are independent, thus, the probability of a certain link failure scenario is the product of individual link failure probabilities. Then, we would want to enforce the following constraint to search for failure scenarios with probability greater than ω:

$$\prod_{l \in Links. Fail(l) = 1} P_{fail}(l) \geq \omega$$

We use the logarithm of probabilities to generate the equivalent linear constraint.

$$\sum_{l \in Links} log(P_{fail}(l)) \times Fail(l) \geq log(\omega) \tag{2.12}$$

We can have a theorem similar to Theorem 2.3.1 for link failure probabilities.

2.4 Verification using QARC

Finally, verifying for network load violations using QARC is done by adding load-related constraints.

$Load(e) = 1$ iff the utilization of link e exceeds capacity.

The following constraint correctly sets the value of the variable $Load(e) \in \{0,1\}$:

$$\frac{\sum_{tc \in TC}[Flow(e,tc) + \Delta(e,tc)] \times T(tc)}{Cap(e)} - Load(e) \geq 0 \tag{2.13}$$

The numerator of the fraction captures the total traffic flow on link e, while the denominator, $Cap(e)$ (a constant), captures the capacity of e. $Load(e)$ can only be 1 when the traffic on link e exceeds its capacity. To find if there exists a failure scenario where at least one of the links is overloaded, we can constrain the sum of load variables to be at least 1:

$$\sum_{e \in Links} Load(e) \geq 1 \tag{2.14}$$

When feeding the constraints to the ILP solver, there can be two outcomes. First, the solver returns a satisfiable model from which we can extract the link failure scenario under which one or more links are overloaded. Second, the solver returns unsatisfiable, which means there is no k-link failure scenario that can cause link overload. There are two explanations for this outcome: (a) the network has sufficient capacity to handle rerouted input traffic under failures, or (b) a subset of traffic classes that do not have high path redundancy are disconnected due to the failures, and the remaining active traffic classes do not have sufficient traffic to cause overloads. Note that, even when the solver finds a failure scenario where link overload occurs, certain traffic classes could be disconnected in the satisfying solution, with the remaining active traffic classes still able to overload the link(s). To summarize, QARC, as presented, will try to find potential for link overload even in the face of disconnections.

Theorem 2.4.1 (Correctness)**.** *A failure scenario FL satisfies Constraints 2.1-2.14 if and only if there exists a link $l \in Links$ such that $Util(l, FL) \geq Cap(l)$.*

2.5 Optimizations

We now describe two techniques that take advantage of the properties of QARC abstraction and constraints to speed up verification performance. First, we use ARC's shortest-distance path property to efficiently minimize the traffic class ETGs (§2.5.1). Second, we show how to use minimized ETGs to partition the set of network links and perform verification in a parallel fashion (§2.5.2).

2.5.1 ETG Minimization

Due to Theorem 2.2.1, in an ETG, the traffic will not traverse a path with a higher cost if one with a lower cost exists. Therefore, for each traffic class, we can prune the ETG and remove edges and nodes that will not be traversed under the targeted number of failures. This pruning strategy reduces the size of the generated constraints and can result in faster verification.

To illustrate this property, consider the network in Figure 2.4. For any 1-link failure scenario, the traffic from S to T will never traverse router C on links $B \rightarrow C$ and $C \rightarrow A$ (because 2 shorter paths exist in the network). Thus, in the ETG, we can prune all nodes and edges corresponding to router C, forming a minimized ETG.

Identifying the redundant nodes and edges for general k link failures can be challenging—the naive approach of enumerating all k-link failures will be expensive. Instead, we modify QARC's encoding to perform this task. The core insight of our approach is to find the maximum shortest distance between the source and the destination under any k-link failure scenario—all nodes and edges that are farther than such a distance will never be traversed and can be pruned from the ETG. The same minimization cannot be performed efficiently when using Minesweeper's symbolic SMT-based abstraction: one would need to check each node and edge to detect if the node/edge is reachable under k failures. In ARC, since the edge weights are not symbolic, we can eliminate a path if "shorter paths" exist.

Minimization Constraints. Unlike verification which deals with all traffic classes, minimization only involves constraints for a single traffic class. The minimization of multiple ETGs can be done in parallel since one ETG's routing does not depend on other ETGs. For an ETG of traffic class tc, we use Constraints (2.5) to specify upper bounds on distances and Constraints (2.11) to bound the number of failures. The maximum shortest distance between the source and destination of the ETG can be found using the objective:

$$\texttt{maximize } \mathit{Dist}(\mathit{Src}(tc), tc) \tag{2.15}$$

Let us denote the objective value provided by the ILP solver as *MaxDist*. Given an edge $e : n_1 \rightarrow n_2$, if the shortest path from source to destination using the edge $n_1 \rightarrow n_2$ is greater than *MaxDist*, the edge will never be traversed under any k link failures and can be pruned from the ETG. We check for such edges using the condition:

$$SP(Src(tc), n_1) + W(e, tc) + SP(n_2, Dst(tc)) > MaxDist \tag{2.16}$$

where $SP(n, n')$ denotes the length of the shortest path between n and n' in the graph (computed using Dijkstra's shortest path algorithm). After pruning the edges that are never traversed, we prune the nodes that do not have incoming or outgoing edges to obtain a minimized ETG $ETG_{min}(tc)$. ETG minimization is sound and complete, i.e., it will remove edges where traffic will never flow under k failures and will not remove edges where traffic could flow under some k failure scenario.

2.5.2 Parallel Verification

A major bottleneck of QARC verification is solving a Mixed Integer Linear Program (MIP) that has size linear in the number of network links and traffic classes. Consider the network-wide constraint (2.14) used to find load violations where at least one link's utilization exceeds its capacity:

$$\sum_{e \in Links} Load(e) \geq 1$$

If we restrict the above constraint to only consider a subset of links, QARC will find scenarios where one of these links' utilization exceeds its capacity. Thus, if we partition the set of links into N partitions, we can run N instances of verification on separate machines. By splitting the verification N-way, each instance will effectively have a smaller search space of links to find load violations, thus, speeding up verification.

There are numerous ways to partition the set of links. Instead of simply splitting the set of links into N equal-sized partitions, we leverage ETG minimization to generate a partitioning scheme that also minimizes the number of the traffic classes that contribute to load on links in each partition. The insight of our partitioning scheme is as follows: a minimized ETG prunes all links where traffic for the particular class does not flow under any k-link failure scenario. To detect if link e can be overloaded, we can prune the constraints for traffic classes that do not contain e in their minimized ETGs. Thus, for each partition, we only need to consider a subset of traffic classes, reducing verification time further.

We formulate our partitioning scheme as an Integer Linear Program (ILP). Our partitioning scheme is tied inherently to the ETG minimization, which we can perform efficiently

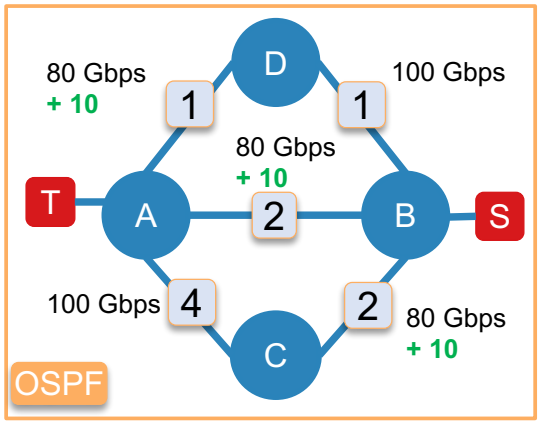


Fig. 2.4 Example OSPF network where a link gets overloaded under 1 and 2 link failure scenarios. The capacities in green are the minimum link capacity additions computed by QARC to ensure no violations occur under ≤ 2 failures.

due to the ARC abstraction. Without ETG minimization, all traffic classes would contain all links in their ETG (barring ACLs and filters), thus, we would not be able to further reduce the constraints in each partition.

2.6 Upgrading Link Capacities in QARC

Once a load violation is found, the operator might want to modify the network to ensure that all link utilizations do not exceed capacity under all considered failure scenarios. One way to achieve this property is to modify the control plane. However, modifying the control plane may cause violations of *qualitative* properties that the current configuration satisfies (access control, waypointing for middleboxes, etc.). Finding policy-compliant control-plane modifications is already a computationally hard problem [36], and accounting for network load on all links for all traffic classes complicates the problem further.

Instead, we propose to use the QARC abstraction to *find new link capacities guaranteeing all link utilizations do not exceed capacity under all considered failure scenarios*. There are multiple mechanisms for realizing the new capacities—physically adding more cables to increase capacity, or dynamically changing capacity in the case of optical links [80]. Increasing bandwidth is expensive, thus, we need to ensure we do not increase capacities beyond what is necessary.

Upgrade Formulation. Concretely, the upgrade problem is as follows: *Given a set of network configurations and input characteristics, find the minimal set of links and the minimal capacity additions to these links such that the network is not overloaded under the given failure scenarios.*

Figure 2.4 illustrates the upgraded link capacities computed by QARC. The traffic sent by class $S \rightarrow T$ is 90 Gbps. As we can see, under 2-link failure scenarios (e.g., $D-A$ and $B-A$), some links' utilization exceeds capacity (e.g., $B-C$). QARC computes the minimum capacity additions to the links (shown in green—e.g., +10); by increasing the capacities, no 1- or 2-link failure scenario can cause violations.

We use QARC to compute new link capacities such that link utilization never exceeds the "new" capacity under the provided failure scenarios. We use the QARC constraints (§2.3.1-§2.3.4) and compute the minimum link capacity that link l should have using the following objective:

$$\texttt{maximize} \sum_{tc \in TC} [Flow(l,tc) + \Delta(l,tc)] \times T(tc) \tag{2.17}$$

QARC computes the maximum utilization for the link under the different failure scenarios given the input traffic characteristics with bounded variation, which is the *minimum new capacity* required on the link to ensure that the particular link is not overloaded. Note that, this capacity is not dependent on the capacity of other links, as the distributed control plane chooses the "best" active path(s) under failures based on configuration parameters such as static link weights and path lengths. Thus, similar to verification, we can parallelize the phase of computing the upgraded link capacities.

The set of links whose capacities need an upgrade is *minimal*—i.e., if any of these links have capacities less than the computed ones, there exists a failure scenario under which a link will be overloaded.

2.7 Limitations

QARC relies on the underlying ARC's graph abstraction and the path-equivalence property to model the flow of traffic in the network to detect load violations. Thus, QARC can only be used for network configurations that can be faithfully represented by ARC. ARC cannot model local path selection criteria, e.g., BGP local preference, because it is not possible to statically assign edge weights to ETGs such that local decisions (e.g., local preferences) override the global costs (e.g., path lengths) without compromising the path equivalence property. Similarly, iBGP's path selection does not use global costs like path lengths, and hence, cannot be modeled by ARC. ARC also does not support BGP communities, which are tags on route advertisements that influence path selection at routers. While state-of-art frameworks like Minesweeper [10] can support the above configuration constructs, the generality comes at the cost of performance (Figure 2.2). Tiramisu [7] also supports a wide

range of configuration constructs like Minesweeper, but Tiramisu's abstraction is tied to qualitative verification algorithms, and cannot be extended to support quantitative verification.

Moreover, a majority of real-world network configurations do not deploy these complex BGP features. Gember-Jacobson et al. [37] analyzed the network configurations of 314 datacenter networks operated by a large Online Service Provider and showed that ARC produces path-equivalent abstractions for more than 95% of the networks. Those networks did not use BGP local preferences or other protocol features not supported by ARC. Thus, QARC has sufficiently high feature coverage to be useful in real-world settings.

2.8 Evaluation

QARC is implemented in Java using the open-source ARC [6] and Batfish [5] frameworks. QARC uses the Gurobi ILP solver [46] for solving the verification and repair constraints. Our evaluation answers the following questions:

Q1: Do real datacenter and ISP networks suffer from link overload under failures? (§2.8.1)

Q2: How quickly can QARC detect load violations or report the absence of violations? (§2.8.2)

Q3: Do our optimizations speed up verification? (§2.8.3)

Q4: How quickly can QARC compute links to upgrade? (§2.8.4)

Experiments were conducted on a 5-node 40-core Intel-Xeon 2.40GHz CPU machine with 128GB of RAM.

Networks. We study 112 datacenter networks operated by Microsoft, and 86 ISP networks obtained from the Topology Zoo [54] dataset. The network configurations use OSPF, BGP, static routing constructs, and ECMP load-balancing which are all modeled by ARC. We refer to the datacenter networks as DC*i*, and the ISP topologies by their names from Topology Zoo. We also study two synthetic fat-tree [8] datacenter topologies: Fat6 (45 routers) and Fat8 (80 routers).

Link Capacities. We vary the link capacities to be either 40Gbps or 100Gbps picked randomly or dependent on topology structure (e.g., for fat-tree: core-aggregate links are 100Gbps while edge-aggregate links are 40Gbps).

Traffic Matrices (TM). To verify the datacenter networks on realistic input traffic characteristics, we sample the datacenter packet traces from the Fbflow dataset [76]. We use the gravity [75] model to generate traffic matrices for our datacenter and ISP networks. The traffic matrices describe the traffic between all edge-edge router pairs for the datacenter networks or all endpoint pairs for the ISP networks. We denote the maximum link utilization of our network with 0 failures as MLU_0. We run our experiments using the traffic matrices

Table 2.3 Networks which experience link overload for one or more TMs (>0%) and more than 10 of 20 TMs (>50%) at $k = 1$ and $k = 2$.

Class	Total	k=1 (>0%)	k=1 (>50%)	k=2 (>0%)	k=2 (>50%)
DC	114	56	20	52	23
ISP	86	83	67	82	63

Table 2.4 Percentage overload violations for 1-link failures for datacenter networks with Fbflow and Gravity traffic matrices

Matrix	DC1 (<100)	DC2 (<100)	DC3 (<100)	Fat6 (216)	DC4 (<1000)
Fbflow	74%	72%	28%	100%	100%
Gravity	83%	96%	38%	100%	100%

obtained from Fbflow or gravity which have the property that $MLU_0 \in [0.65, 0.8]$ [1]; under no failures, the max link utilization in the network is less than 80%. Our traffic matrices model scenarios when the network is not overloaded when all links are active, but certain 1 or 2 link failures could potentially lead to some links getting overloaded. Unless otherwise mentioned, we bound the variation of an individual traffic class so that it cannot increase more than 50%, and we bound the total traffic variation summed over all traffic classes to 10% of the total network traffic.

2.8.1 Verifying Real Networks

We use QARC to detect if the datacenter (DC) and ISP networks experience link overload under $k = \{1, 2\}$ link failures. We generate 20 random traffic matrices (Fbflow for datacenter and gravity for ISP) and run QARC's verification. We report our findings in Table 2.3.

QARC finds link overload events for 1-link failures for 139 of the 200 networks. Moreover, 87 networks show link overload violations for more than 10 out of 20 TMs, indicating that these networks are quite susceptible to link overload events even under a single-link failure scenario. QARC likewise finds overload for over 90% of the ISP networks, compared to 50% for the datacenter networks, indicating that datacenter networks are less susceptible to link overload due to higher path diversity than ISP networks. Notice that, the number of networks experiencing violations decreases for 2-link compared to 1-link failures. This is because 2-link failures disconnect more traffic classes; the number of active traffic classes reduces, and thus, links' utilizations do not exceed capacities.

Min Traffic Variation. QARC can be used to find the minimum total variation of traffic (each traffic class cannot increase by more than 50%) which can cause overload under 1-link

[1] We extract matrices using gravity/fbflow and multiply each field of the matrix with a constant factor to obtain the desired MLU_0

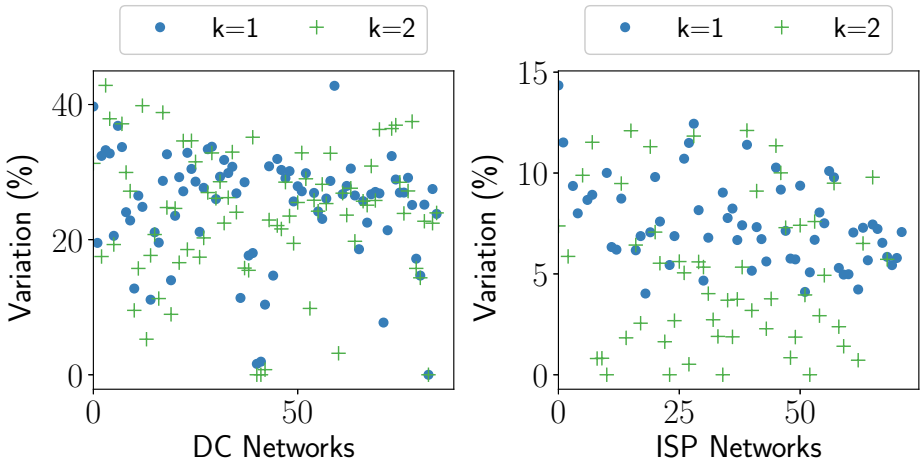
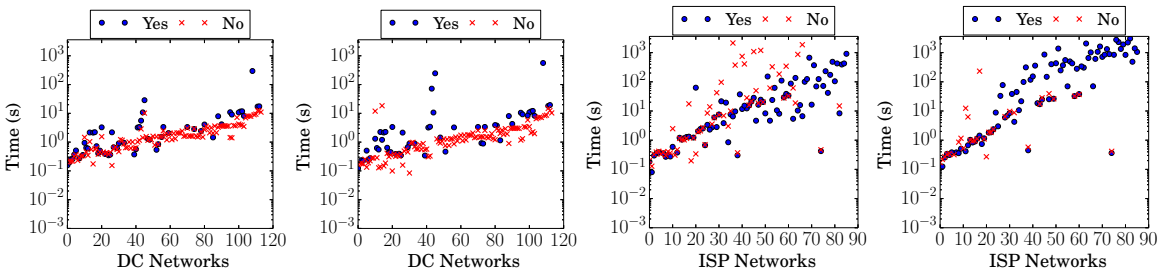


Fig. 2.5 Variation % for different ISP and datacenter networks for 1 and 2 link failure scenarios.

and 2-link failure scenarios. Minimum variation can be used to judge how the network handles unexpected spikes in load under failures. For this experiment, we use traffic matrices such that $MLU_0 = 0.7$ (high utilization) and find the average minimum traffic variation. For our ISP and datacenter networks in Figure 2.5, we report the variation as a percentage of total network volume. We can observe that the ISP networks require lower variations (2-12%) to cause link overload, i.e., if the total traffic increases by 12%, the network is likely to be overloaded under failures. Meanwhile, datacenter networks require more traffic variation for links to get overloaded (2-40%), highlighting intrinsic robustness pertaining to meeting traffic demands and higher path diversity. Also, $k = 2$ requires lower variation than $k = 1$.

Effect of Traffic Matrices. We also study the effect of different types of traffic matrices causing link overloads. We consider 5 datacenter networks and use QARC to find link overload for $k = 1$ failure scenarios. We generate 100 TMs using both Fbflow and gravity models such that $MLU_0 = [0.65, 0.8]$ and bounded variation is 10%; in Table 2.4 we report the percentage of violations found in these networks for the different matrices. We observe that Fbflow matrices lead to fewer load violations than gravity matrices for some of the datacenter networks. This shows the impact of using QARC to verify if significant changes to traffic patterns can affect a network's load properties under failures.

Q1: QARC **finds network load violations under failures** in 70% of real datacenter and ISP networks, and ISP networks are more susceptible to link overload than datacenter networks. To the best of our knowledge, this is the first study of finding potential violations for control planes.



(a) k=1 (b) k=2 (c) k=1 (d) k=2

Fig. 2.6 Verification time (log scale) sorted by network links for different WAN and datacenter networks for 1 and 2 link failure scenarios.

2.8.2 Verification Performance

We now evaluate the performance of QARC's algorithm for detecting link overload for the datacenter and ISP networks using 20 different traffic matrices with $MLU_0 \in [0.65, 0.8]$ for 1 and 2-link failure scenarios. The datacenter networks have on the order of 10 to order of 100 links (exact numbers hidden for confidentiality) and order of 100 traffic classes, and the ISP networks have 8-125 links and 10-2000 traffic classes. We use 5 nodes to run parallel verification and report the average time to successfully find violations (Yes) or verify no violations (No) for the networks.

Figure 2.6 shows the results. We observe that verification time for $k = 2$ is greater than $k = 1$ due to the larger search space. We sort the networks by number of network links and observe that network time in general increases with size of the network. Also, the ISP topologies have higher verification times compared to the datacenter networks (due to larger number of traffic classes). For most networks, QARC is able to verify if the network will experience link overload (Yes) or not (No) in under an hour (6.7% of the runs timed out due to the large number of links or traffic classes). With our choice of traffic matrices with $MLU_0 \in [0.65, 0.8]$, the times taken to find a violation or proving the absence of violations follow similar trends, indicating similar difficulty.

We also look at QARC's performance with increasing number of link failures. We consider 3 datacenter networks (DC2, DC4, Fat6) and one ISP network (Abilene) and use 50 random gravity matrices with $MLU_0 \in [0.65, 0.8]$ and 10% traffic variation, and verify for $k = [1, 4]$ failure scenarios. We present the median verification times in Table 2.5. As k increases, verification times generally increase (Fat-6 and DC4 experience a drastic increase at $k = 3$ and $k = 4$, respectively). For $k \leq 2$, QARC terminates within an hour for most instances (<1% instances timed out). QARC experiences more timeouts for $k = 3$ (7%) and $k = 4$ (25%) due to the exponential complexity of verification.

Table 2.5 Verification times (s) for $k = [1,4]$ link failures.

Network (Links)	k=1	k=2	k=3	k=4
Abilene (28)	0.7	7.9	9.1	10.7
DC2 (<100)	1.4	17.5	17.1	20.1
Fat6 (216)	3.3	2.8	1,550.6	1,357.7
DC4 (<1000)	9.6	22.1	81.2	1,496.0

We now consider how QARC performs compared to naive enumeration. We fix the input traffic matrix, find the time taken to verify one failure and extrapolate by the number of failures. We consider a 5-node parallelism, so enumeration is spread across 5 machines equally. For a network with >100 links, the naive enumeration time versus median QARC time for different failure scenarios are: (a) $k = 1$: 12.8s vs 9.6s, (b) $k = 2$: 1,017s vs 22s, (c) $k = 3$: 14.8hrs vs 81s, and (d) $k = 4$: 584hrs vs 0.4hrs. While for 1-link failures, the naive enumeration is comparable, QARC has significant speedup for $k \geq 2$. Moreover, verification with non-zero traffic matrix variation cannot be performed by enumeration, as there are infinitely many rational-value traffic matrices to consider.

Q2: QARC **can verify network overload for medium-sized datacenter and ISP networks** for different failure scenarios in under an hour.

2.8.3 QARC Optimizations

We now evaluate how the optimizations presented in §2.5 improve QARC's performance. We consider 3 datacenter (DC2, DC4, Fat6) and one ISP network (Abilene) and use 50 random gravity matrices with $MLU_0 \in [0.65, 0.8]$ and no traffic variation, and verify for $k = 1$ failures.

ETG Minimization. Table 2.6 reports the speedup obtained by ETG minimization (speedup = time without minimization/ time with minimization) and edge reductions for the networks. For the bigger networks, the speedup in verification is significantly larger, aiding in making QARC's verification more tractable. When verifying the Fat6 and DC4 network with $k = 1$, we are able to reduce 86% and 95% of the ETG edges respectively, achieving significant speedups of 69× and 18× for Fat6 and DC4. The ETG minimization phase is quick, taking under 5 seconds for all the networks.

Parallelization. We now consider the speedup achieved by parallelizing the verification (with optimal partitions) over verification run on a single node. We run verification on 5 nodes and compare the performance with 1-node verification. In the parallel scenario, we terminate verification if any one node finds a link load violation, otherwise we wait until all nodes report no violations (with a timeout of 1 hour). Table 2.6 shows the verification speedup achieved due to parallelization. For the bigger networks (Fat6 and DC4), we achieve

Table 2.6 Speedups due to optimizations for $k = 1$ failures.

Optimizations	**Abil-ene(28)**	**DC2 (<100)**	**Fat6 (216)**	**DC4 (<1000)**
Edges Removed	54%	67%	86%	95%
Minimize Speedup	2.3	4.5	69	18
Parallel Speedup	1.3	1.3	5.0	9.5
Partition Classes	68%	50%	42%	36%
Partition Speedup	1.00	1.19	1.38	1.52

average speedup of over 5× and 10× over the single node case; thus, QARC parallelization further improves the tractability of verification.

Partitioning. Finally, we evaluate the speedup due to the optimal partitioning scheme which minimizes the number of traffic classes in each of the 5 partitions. We compare the performance of 5-node optimal partition verification to to a 5-node random partitioning scheme which randomly divides the links in equal sized partitions. For this experiment, we pre-compute the optimal partitioning (not included in verification time) and compare verification using the optimal and random naive partitions on 5 nodes. Table 2.6 shows the verification speedup achieved by the optimal versus naive partitions. The speedup achieved for DC2, DC4 and Fat6 is about 20-50%. We report the percentage of traffic classes in the optimal partitions (Partition Classes) for each network in Table 2.6; we are able to reduce more than half of the traffic classes in each partition for DC2, DC4 and Fat6.

Q3: QARC **optimizations speed up verification** significantly for different networks and failure scenarios, with the most benefits coming from ETG minimization and parallel verification for the bigger Fat6 and DC4 networks.

2.8.4 Upgrade Performance

We use QARC to generate the minimal link capacity additions required to prevent link overload under 1-link and 2-link failures. For this experiment, we consider traffic matrices with $MLU_0 \in [0.8, 0.95]$ and zero variation. Similar to verification, we parallelize the phase of computing the new link capacities to 5 nodes. We report the median upgrade time for 5 different networks in Figure 2.7. QARC is able to compute new link capacities for the networks in under 5 minutes using the 5 nodes. In our runs, QARC requires changing capacities of order of 10 links (Fat6 repair has the highest number of links whose capacity has to be increased). Finding the repair for all 2-link failure scenarios takes more time than 1-link scenarios due to the larger search space.

Q3: QARC **can upgrade link capacities to prevent network overload under failures** for different networks in under 400 seconds.

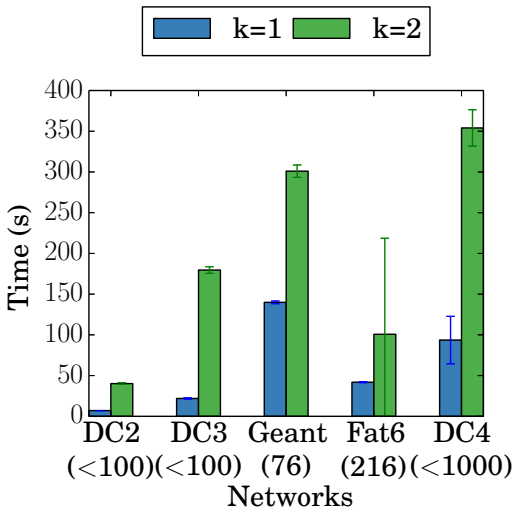


Fig. 2.7 Capacity upgrade computation time

2.9 Other Related Work

Config. verification/repair. Our work complements configuration analysis tools [32, 37, 31, 10, 42, 7, 92, 90, 11, 87, 29, 36] that focus on qualitative properties; to the best of our knowledge, we are the first to show how to reason about quantitative properties in networks with distributed control planes. Our ETG minimization approach is inspired by the idea of surgery proposed by Plotkin et al. [69] to slice the network and headers to speed up reachability verification. The key difference is that QARC identifies network components which are not traversed under different failure scenarios, while Plotkin et al. do not verify reachability under failures.

Quantitative properties. ProbNetKAT [35, 82] supports probabilistic routing behavior and can answer congestion and latency queries about the network using a link failure probability distribution. ProbNetKAT cannot express complex routing strategies—e.g., distributed routing protocols such as OSPF—that recompute new paths under failures based on the global network state. Chang et al.. [21] propose a framework to validate network design under failures and uncertain demands. Specifically, the input is a global routing strategy that can adapt to the current network topology to optimize the max link utilization (MLU); examples are multi-commodity flow (which finds optimal network paths) and MPLS tunneling (best path chosen from a set of pre-specified tunnels). The system finds the worst case (max) value that MLU can achieve under a set of failure scenarios/traffic demands when the adaptive routing strategy is in use by relaxing a non-linear formulation to a linear program.

The routing in distributed control planes does not seek to minimize the global objective of maximum link utilization (MLU), and, thus, cannot be reasoned about directly using this framework. QARC leverages ARC to reason about real router configurations deployed

in different real-world networks, while Chang et al.'s framework deals with the abstract routing strategies like multi-commodity flow. Moreover, our formulation is sound, i.e., it will find a failure scenario where link overload happens if one exists, however Chang et al.'s framework solves a relaxed formulation, so cannot provide soundness guarantees for all routing strategies.

Traffic Engineering. Our work is orthogonal to traffic engineering (TE) works [33, 91, 56, 57] that develop/configure routing strategies to react to uncertain traffic demands or failure scenarios. In a sense, TE systems solve a narrow "synthesis" problem, whereas we focus on analyzing the joint impact of routing, input traffic characteristics, and arbitrary failures on the network's current link bandwidths, i.e., *verification*. Also, the scope of TE systems is narrow in the sense that they only consider generating routes that satisfy quantitative properties, but ignore effects of such routes on qualitative properties.

Chapter 3

Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks

Many enterprises are increasingly migrating their on-premise IT infrastructure to cloud datacenters. In such environments, the different enterprises (tenants) share different resources, such as, the compute machines that run their applications and network infrastructure used for communication among these applications. Operators of such multi-tenant datacenters thus have to deal with a multitude of machines communicating with each other (flows) over a network that is composed of many tens to hundreds of routers or switches (devices) [41]. With growing diversity of enterprise applications and the need for security and compliance, these pathways of communication through the datacenter network are subject to increasingly complex network-based policies.

Consider a tenant in such a datacenter. She may desire basic communication among her applications (reachability) along shortest paths based on certain metrics. In addition, she may wish that traffic attempting to reach some of her applications is examined by a set of "middleboxes" (traversal) for auditing and access control. For strong security or Quality-of-Service considerations, a tenant may additionally desire that a subset of her flows does not share any infrastructure with others' flows (isolation). In parallel, cloud operators must meet key operational policies. For instance, they often need to optimize network performance objectives (traffic engineering), e.g., minimizing the maximum load imposed by all tenants on network links, and deal with resource constraints such as link capacity bounds and switch table sizes. Also, since datacenter networks are highly prone to link/switch failures [43], operators need to gracefully transition the old (pre-failure) data plane to a policy-compliant one (post-failure) in a rapid and/or efficient manner.

Today, configuring network devices to enforce these complex policies in aggregate is manual, ad hoc, and error-prone. This can lead to misconfigurations and violations of tenant service-level agreements, which can have severe performance and security impacts.

With software-defined networking (SDN), operators can program networks in a more intuitive manner. In SDN, a general-purpose centralized controller machine (control plane) controls end-to-end communication pathways by managing network forwarding rules on a collection of programmable switches (data plane). Using a global view of the current network topology, the controller can program forwarding rules on switches based on application requirements. Unfortunately, many existing SDN programming languages [34, 64] present too narrow a view: operators would ideally want to specify and realize policies network-wide, whereas these languages focus on programming *individual* switch behaviors. Other recent works on network-wide policy enforcement [83, 71, 73, 9, 81, 48] go beyond the single-switch model, but they target specific types of policies and are not easily extensible to different kinds of policies. Notably, NetKAT is among the most expressive and can be used to encode certain network-wide policies like regular paths and programs on virtual topologies, however, it cannot be used to express policies based on hyperproperties [26] (where one class's path is dependent on the other) like isolated paths or traffic engineering. Furthermore, for many types of policies, generating a data plane that enforces them is a computationally hard problem, requiring the design of efficient custom heuristics per policy type.

In this work, we seek a general approach that allows a variety of rich policies to be specified as the input, with the output being the corresponding set of switch forwarding rules such that the complexities of correctly realizing the policies in the data plane are hidden from operators. This is an important step toward *intent-based networking* [3], where operators specify *what* they want the network to do instead of worrying about *how* the network must be configured. We argue that data plane synthesis can help realize this vision in the multi-tenant datacenter context.

We present Genesis, a framework for declaratively specifying and enforcing complex policies such as, isolation, middlebox traversals, network optimization objectives, and failure resilience. To tackle the high complexity of enforcing some of these policies (e.g., enforcing isolation is NP-complete), Genesis encodes the problem of enforcing policies as a constraint solving problem and leverages recent advances in fast Satisfiability Modulo Theories (SMT) solvers to efficiently search for a solution to the constraints. The solution is then translated into switch forwarding rules. Genesis uses two intuitive relations that concisely capture the semantics of custom network forwarding behaviors. These help express a variety of both path-based and global policies desired in a datacenter. Interestingly, complex global policies (specifically, policy-compliant failure resilience) can be realized within this

framework without requiring additional encoding (of specific failure scenarios) by just cleverly transforming path-based policies. By leveraging the formal guarantees of constraint solving, Genesis eliminates the room for error in the enforcement of complex policies.

Further, we present two novel techniques that leverage domain-specific properties to speed up Genesis's synthesis. First, Genesis allows the network operator to write restricted forms of regular expressions, called *tactics*, that blacklist paths based on certain patterns that are not desired in a datacenter network (e.g., paths that alternate between topology tiers). These tactics are used to discard several constraints, acting as a search strategy for the solver. Tactics can speed up the synthesis procedure by $1.5 - 400\times$ (median speedup: $1.6\times$, average speedup: $22\times$). Second, we develop a *divide-and-conquer* synthesis procedure that opportunistically leverages the dependency relationships among isolation policies to improve synthesis performance. The procedure partitions the input policies into components such that Genesis can synthesize these components separately and faster than the complete problem. Divide-and-conquer synthesis can halve the synthesis time for 40% of synthetic isolation workloads.

Contributions.

- An extensible declarative framework for describing complex policies like isolation, waypoints (§3.3), traffic engineering (§3.4), and failure resiliency (§3.5.1) and a modular SMT-based algorithm for enforcing such policies;
- A tactic-based synthesis algorithm, which leverages datacenter network structure to blacklist undesirable path patterns (§3.6);
- A divide-and-conquer procedure for speeding up synthesis by leveraging the structure of policy interactions (§3.7);
- An implementation of Genesis and an evaluation on different policy workloads, topologies and multi-tenancy settings (§3.8).

3.1 Preliminaries and Policies Supported

We describe the type of policies desired in multi-tenant data centers that Genesis supports. We use Figure 3.1 as a running example. In our setting, an "operator" manages a multi-tenant datacenter like an enterprise network or a private datacenter. The operator specifies "operator policies" which reflect important global objectives pertaining to how she wishes to manage her overall infrastructure. A tenant is an entity (e.g., an enterprise or a department thereof) that has offloaded its IT infrastructure to the datacenter. Each tenant controls a number of host machines in the datacenter running some of its applications, and specifies path-based policies (as opposed to operator's global policies). Tenant policies define whether paths can

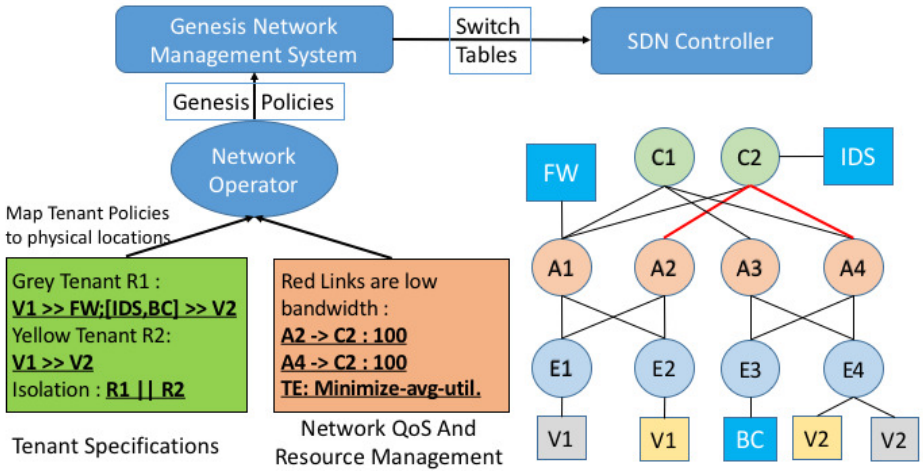


Fig. 3.1 Genesis in a multi-tenant datacenter setting of a network containing several VMs and middleboxes. The network operator translates the tenant specifications and network resource policies to Genesis policies and Genesis synthesizes switch forwarding rules which are installed by the SDN controller.

exist among its hosts, and if so, what additional properties the paths must satisfy for security, performance or access control reasons. Given the policies, the cloud operator solves the VM placement problem separately[1]. The resulting tenant VM locations, along with the policies to apply on paths between locations are then provided as input to Genesis. Tenants are *unaware* of the physical topology and cannot program physical switches directly, maintaining the virtual network abstraction.

Figure 3.1 shows several tenants who differ in the nature of policies they wish to realize; we also show the operator policies. The policies supported by Genesis are described below. Notice that these reflect and, in some cases, extend policies that today's enterprises and datacenter operators realize in their networks [41].

Tenant Policy: Reachability. This enables network communication between specific pairs of tenant's virtual instances (VMs), applications, or hosts. In our example, one tenant has defined a reachability policy (R2) for its two VMs: `V1 » V2`, which is translated after VM placement to the source and destination switches `E2 » E4`. A pair of VMs, applications or hosts that are allowed to communicate by means of a reachability policy defines a flow or a "packet class"; we use these terms interchangeably. Any communication that is not defined by a reachability policy is implicitly blocked (i.e., all communication is "default off").

Tenant Policy: Middlebox Traversals. A tenant may wish that the flow between two of her end hosts, or from another tenant, must traverse specific middleboxes, which we also refer to as "waypoints" in this work. Middleboxes are custom processing appliances often used for security, access control, or performance reasons (e.g., firewalls, intrusion prevention

[1] Genesis currently does not help in the VM placement problem.

Type	Policy	GPL Syntax	Description
Tenant	Reachability	`predicate : src » dst`	Forwarding Rules for path from switch *src* to switch *dst* for packets matching *predicate*
	Reachability with Ordered Waypoints	`predicate : src » W1; ...;Wn » dst`	Forwarding Rules for path from switch *src* to switch *dst* for packets matching *predicate* such that the path traverses $w \in W_1$ in any order, then $w \in W_2$ in any order after all waypoints in W_1 are traversed and so on.
	Traffic Isolation	`R1 || R2`	Paths of two reachability policies $R1$ and $R2$ do not share a link in the same direction
	Link Isolation	`R1 <> R2`	Paths of two reachability policies $R1$ and $R2$ do not share a link in any direction (edge-disjoint)
Operator	Link Capacity	$sw_1 \rightarrow sw_2$: capacity	The weights of flows traversing the link $sw_1 \rightarrow sw_2$ do not exceed *capacity*
	Switch Table Size	`sw : size`	The number of flows traversing through *sw* do not exceed *size* as each flow would require a forwarding rule at *sw*
	Traffic Engineering	`minimize-tot-te`, `minimize-max-te`	TE objectives: minimize total/max link utilization

Table 3.1 Genesis Policy Support with Genesis Policy Language (GPL) syntax

systems, monitoring/accounting gateways, proxies, and load balancers). Specifically, for particular flows of interest, a tenant can provide a sequence of unordered sets of middleboxes to traverse [70]. The flows must traverse these sets in order, while in a set, all middleboxes must be traversed and the order is irrelevant.[2] For example, one of the tenants defines a traversal policy (R1): `V1 » FW; [IDS,BC] » V2` which specifies that traffic must first pass through the firewall (FW), and then through the Intrusion detection system (IDS) and the byte counter (BC) in any order.

Tenant Policy: Isolation. Tenants may require various Quality-of-Service (QoS) or security guarantees that stipulate varying degrees of isolation for their traffic. In the extreme, a tenant could require that her flows are not affected in any manner by any other tenant by strictly isolating the path of the tenant's flows from others' flows. In Figure 3.1, we have two tenants whose traffic will be isolated from one another (`R1||R2`), i.e., the network paths used by the tenants will not share any links in the topology. A tenant could also specify isolation for a subset of her (performance-sensitive) flows from other flows of the same tenant or those belonging to other tenants; the rest of the tenant's flows may require no guarantees.

[2]The unordered set abstraction leverages the fact that middleboxes without dependencies in their traffic processing behavior can be placed in any order relative to each other [70].

Operator Policy: Managing Capacity Constraints. While support for the above policies can be used to satisfy tenant requirements, network operators often wish to carefully manage constrained resources. Common examples that Genesis supports include enforcing strict constraints on aggregate number of flows traversing a switch (due to all tenants) so as to adhere to switch memory constraints, and ensuring that the total load on certain links is within predefined thresholds (we assume here that each flow has a predefined load that it imposes on the path it uses). In our example in Figure 3.1, A2-C2 and A4-C2 links are of low bandwidth, and the operator wants to ensure that total load on these links does not exceed 100 (`A2` $\rightarrow$ `C2: 100, A4` $\rightarrow$ `C2:100`). These policies could also be used to provide QoS guarantees to tenants like minimum bandwidth guarantees.

Operator Policy: Traffic Engineering. As an alternative to managing strict (link) capacity constraints, operators may also want to balance load on their network infrastructure. This is often done by optimizing a network-wide objective such as total or maximum utilization of network links due to traffic induced by all tenants.

Operator Policy: Handling failure gracefully. Modern networks experience link and switch failures frequently [43]. When a failure occurs, we must reconfigure the forwarding rules so that the policies are satisfied. Naively recomputing forwarding rules incurs an unduly large overhead because old forwarding rules have to be torn down at all switches, and new rules must be installed. Switch rules deletions/insertions take a non-trivial amount of time [47], potentially leading to disruptions. It is therefore desirable to have graceful approaches that either minimize the potential for disruption by minimizing the number of forwarding rules or switches modified in transitioning from an old data-plane, or eliminate the possibility of disruption altogether by precomputing backup policy-compliant paths for a fixed number of failures (at the cost of storing extra rules at switches).

Realizing these policies is challenging today. In particular, state-of-the-art SDN frameworks, e.g., Pyretic [64] and Frenetic [34], are insufficient to program networks to realize them. This is because the above policies are global and cannot be enforced (at least not in an intuitive manner) by programming individual behavior of switches. While some existing SDN-based network management systems [71, 83, 52] overcome these limitations by taking a network-wide view, they are tailored to support specific policies such as middlebox placement or link capacity constraints. As such they cannot enforce several of the policies above.

3.2 Data Plane Synthesis

Our contribution is Genesis, a new general network management system that supports the above policies, and can be extended to support others. The architecture of Genesis is shown

in Figure 3.1. Genesis performs *synthesis* of switch forwarding rules to enforce policies. The policies are specified using Genesis Policy Language, or GPL, as shown in Table 3.1.

Unlike previous efforts in the network synthesis space [77, 83], Genesis is not tailored to specific formalisms such as regular expressions; this aspect makes it *modular* and *easy to extend*. To draw an analogy with SMT solvers, Genesis can be seen as a constraint solver that allows the addition of different types of policies (respectively, theories in SMT) and the design of optimizations based on properties desired by network operators.

Our work is motivated by recent advances in program synthesis, i.e., the task of discovering an executable program from user intent expressed in the form of some constraints. There are three key dimensions to a synthesis problem: the type of constraints that it accepts as expression of user intent, the space of programs over which it searches, and the search technique it employs. Genesis leverages synthesis as follows: given a set of policies which describe tenant and operator intent, the search space is the space of all data planes (i.e., the set of forwarding rules) and the search technique involved is SAT/SMT solving.

Genesis's approach has the following salient features:

(1) Enforcement of the different policies can be translated to the following problem: Given a set of node pairs (derived from the reachability policies) in the graph (topology), find paths in the graph for each of the node pairs satisfying certain properties (derived from the rest of the policies). Thus, the different policies can be enforced by a correct set of forwarding rules at the switches. No extra functionality is required from the controller; its only role is to install the forwarding rules on switches.

(2) Correct enforcement is challenging due to different goals for each of the policies — ensuring isolation between paths may lead to overshooting link utilizations and vice-versa — and is a common cause of incorrect configurations in networks. Our approach removes the need for a verification step in which the operator has to "check" whether the forwarding rules satisfy the desired policies. By using a formal reasoning technique, we are able to consider the space of all data planes and find a solution which is *correct by construction*, eliminating room for operator errors.

(3) Automatically enforcing policies is a task with *high theoretical complexity*. For example, enforcing isolation policies is as hard as solving graph-coloring, an NP-complete problem. Specialized techniques can be used to find the forwarding rules when handling a particular class of policy, but devising good search techniques becomes challenging when multiple types of policies are combined,–e.g., isolation, waypoints, and traffic engineering. Thanks to the many engineering efforts, SMT solvers abstract away most of this complexity and allow us to unify search objectives for every policy into a generalized search technique.

Crucially, Genesis can be extended with ease to support new policies without requiring changes to the underlying search techniques.

In the next subsection, we describe the Genesis synthesis algorithm for tenant policies. We then describe how to accommodate operator policies pertaining to capacity constraints, traffic engineering and failure resiliency (§3.4). Finally, we describe two novel techniques aimed at speeding up Genesis's synthesis: tactics (§3.6) and divide-and-conquer synthesis (§3.7).

3.3 Synthesis of Tenant Policies

The problem statement here is as follows: Given the network topology graph and the set of tenant policies written in GPL, generate paths in the network for every source-destination pair (derived from reachability policies) satisfying all policies. To achieve this, Genesis creates constraints that encode the forwarding and reachability rules pertaining to the paths such that they satisfy the input policies. The synthesized solution of paths obtained from the constraints are then translated to switch forwarding rules.

3.3.1 Network Forwarding Model

We start by describing the basic forwarding model we use in Genesis. We define the physical switch topology as an undirected graph $T = (S, L)$, where S is the set of switches and L is the set of links. We use the neighbour function $N(s) = \{s' \mid (s, s') \in L\}$ to denote the set of neighbour switches of s. We assume a set of packet classes $PC : [0, \lambda]$ and map each reachability policy to a unique integer in PC. In the rest of the thesis, we often use the term packet class to identify the corresponding reachability policy. Other policies are not mapped to packet classes as they do not produce a path, but specify restrictions on paths of packet classes. We use R to denote the set of reachability policies; each policy $r \in R$ is a pair $(predicate{:}src >> W_1; W_2; \ldots; W_n >> dst, pc)$ where:

- $predicate$ is the packet header identifier pertaining to r;
- $src, dst \in S$ are the source and destination switches;
- $W_1, W_2, \ldots, W_n \subseteq S$ are the (potentially empty) ordered sets of waypoints;
- $pc \in PC$ is the packet class and is a unique integer used to identify the variables associated to r

We fix a constant μ and assume all paths have length at most μ. $K = [0, \mu]$ is the set of all permissible path lengths. The network forwarding model abstracts the actual forwarding rules at each node and encodes the reachability of each packet class.

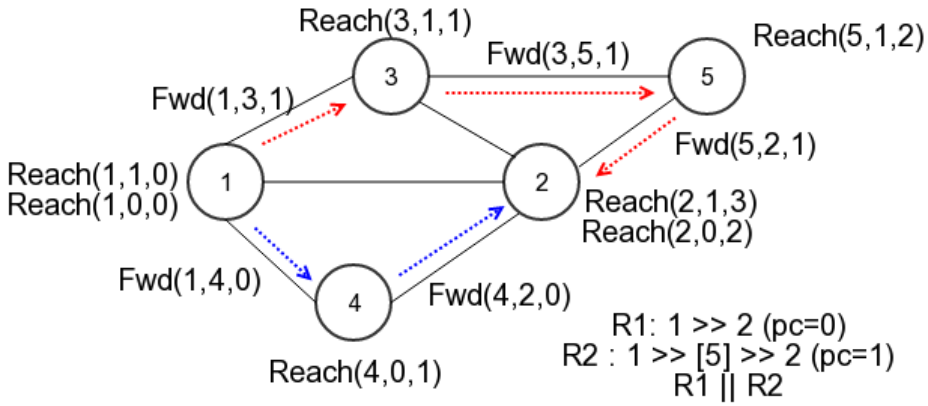


Fig. 3.2 Values of the Fwd and *Reach* relations of the network forwarding model for the policies specified in the figure. The blue and red arrows indicate the paths of packet classes 0 and 1 respectively according to the model.

We use the relation $Fwd \subseteq S \times S \times PC$ to capture the network forwarding behavior—i.e. $(sw_1, sw_2, pc) \in Fwd$ if sw_1 forwards packets of class pc to switch sw_2. We use the relation $Reach \subseteq S \times PC \times K$ to capture the path reachability—i.e. $(sw, pc, k) \in Reach$ if the switch sw is reachable in the path from the source switch of packet class pc in exactly k steps. For brevity, we write $Fwd(sw_1, sw_2, pc)$ for $(sw_1, sw_2, pc) \in Fwd$ and similarly for the *Reach* relation. Since Fwd depends on the topology, for all sw_1, sw_2 that are not connected by a link, we have that $\forall pc, (sw_1, sw_2, pc) \notin Fwd$.

Given a set of policies, Genesis generates a set of constraints denoted by Ψ over the Fwd and *Reach* relations. $(Fwd, Reach) \models \Psi$ denotes that Fwd and *Reach* is a model of Ψ.

Definition 1. *Given two concrete relations Fwd and Reach, the set of induced paths* $\Pi = \texttt{paths}(Fwd, Reach)$ *is defined as follows: given a class pc,* $(sw_0 \ldots sw_k, pc) \in \Pi$ *iff :*

1. $\forall i \in [0,k].(sw_i, pc, i) \in Reach$
2. $\forall i \in [0,k-1].(sw_i, sw_{i+1}, pc) \in Fwd$

Figure 3.2 illustrates these definitions.

Definition 2. *Given the set of constraints* Ψ *corresponding to the input policies, a set of paths* Π *is a solution to* Ψ*,* $\Pi \models \Psi$*, if there exists* $Fwd, Reach$ *such that* $(Fwd, Reach) \models \Psi$ *and* $\Pi = \texttt{paths}(Fwd, Reach)$.

In practice, we model the forwarding and reachability relations using propositions and reduce enforcement of tenant policies like reachability, waypoints and isolation to a Boolean Satisfiability Problem (SAT) problem. Using these relations, operators can write custom policies in a concise and intuitive manner.

3.3.2 Reachability

We first discuss the constraints added to Ψ for reachability policies without waypoints. For a reachability policy $s >> d$ and packet class pc, the added constraints must ensure that the solution model represents a path from source to destination. The base constraint states that $(s, pc, 0) \in Reach$ meaning that s can be reached in 0 steps. The following constraint states that there must be a forwarding rule from s to one of the neighbors of s.[3]

$$\exists n \in N(s).\ Fwd(s,n,pc) \wedge Reach(n,pc,1). \tag{3.1}$$

Next, we add the following constraints that state that d can be reached in some number of steps and, since d is the last switch in the path, there are no forwarding rules from it.

$$\exists k.\ Reach(d,pc,k)\ \wedge\ \forall n \in N(d).\ \neg Fwd(d,n,pc). \tag{3.2}$$

Finally, we add implication constraints that propagate reachability backward from destination to source. If a node n_1 is reachable in k steps, there must be a node n_2 reachable in $k-1$ steps and a forwarding rule $n_2 \rightarrow n_1$.

$$\forall n_1.\forall k \geq 1.\ Reach(n_1,pc,k) \implies \exists n_2.n_2 \in N(n_1) \wedge Reach(n_2,pc,k-1) \wedge Fwd(n_2,n_1,pc). \tag{3.3}$$

When combined together, these constraints are sufficient to ensure the existence of a path from s to d for packet class pc. However, since there is no restriction on the number of Fwd values that can be true at a switch, we can get multiple forwarding rules at switches, and also multiple paths to the destination. These can also create forwarding loops. Concretely, this is not a problem: as long as there is at least one path from s to d we can recover it from the solution of the constraints. Moreover, this representation is quite efficient, as forcing a single path would require adding further constraints (§3.3.3) and increase the synthesis time.

To extract a concrete s-to-d path we perform a breadth-first search on the reachability graph induced by the solution to the constraints. A directed edge $n_1 \rightarrow n_2$ appears in the reachability-graph if there is a forwarding rule indicated by the relation $(n_1,n_2,pc) \in Fwd$. We extract the rules relevant to the shortest path from source to destination from the model, and the additional rules obtained in the solution (extra paths, forwarding loops) are ignored.

[3] We unroll the existential quantifier $\exists n \in N(s)$ using disjunction of clauses $\vee_{n \in N(s)}$ and the universal quantifier $\forall n \in N(dst)$ using conjunction of clauses $\wedge_{n \in N(dst)}$ and stay in propositional logic.

3.3.3 Waypoint Traversal

For a reachability policy with a sequence of waypoint sets $s >> W_1; \ldots; W_n >> d$ and packet class pc, we add all the constraints specified in §3.3.2 to ensure the existence of a path from s to d. We then add constraints so that all waypoints w are traversed.

$$\forall w \in W_1, \ldots, W_n.\ \exists k.\ Reach(w, pc, k). \tag{3.4}$$

For each set W_i for $i > 1$, we add constraints to ensure that all waypoints in W_i are reached after all waypoints in W_{i-1} :

$$\forall w_i \in W_i, \forall k_i.\ Reach(w_i, pc, k_i) \implies \forall w_{i-1} \in W_{i-1}. \\ \exists k_{i-1}.\ k_{i-1} < k_i \wedge Reach(w_{i-1}, pc, k_{i-1}). \tag{3.5}$$

Previously, we imposed no restriction on the number of paths from s to d. In the case of waypoints, this can result in a solution with multiple paths, with each individual path traversing some of the waypoints, which is not the correct enforcement for a waypoint policy. Thus, we need to ensure the solver returns a single path traversing all the waypoints. To achieve this, we limit the number of forwarding rules for pc at a switch to 0 or 1. We define the forwarding set as:

$$FwdSet(sw, pc) = \{k \mid Fwd(sw, k, pc)\}. \tag{3.6}$$

We then add constraints stating that the size of the forwarding set must not exceed 1:

$$\forall sw, pc.\ |FwdSet(sw, pc)| \leq 1. \tag{3.7}$$

Here $|A|$ denotes the size of set A. The above constraints are expressed in SAT as follows:

$$\bigvee_{k_1 \in N(sw)} Fwd(sw, k_1, pc) \wedge \Big(\bigwedge_{k_2 \in N(sw), k_2 \neq k_1} \neg Fwd(sw, k_2, pc)\Big) \tag{3.8}$$

Since, there cannot exist multiple rules at a switch, the model will contain a single path from source to destination for pc traversing the waypoints in the right order.

3.3.4 Isolation

A traffic isolation policy $pc_1 || pc_2$ states that the paths for pc_1 and pc_2 do not share any link in the same direction. We enforce this policy by adding to Ψ, constraints stating that at every

switch, pc_1 and pc_2 must not forward to the same switch:

$$\forall n_1.\ \neg(\exists n_2.Fwd(n_1,n_2,pc_1) \wedge Fwd(n_1,n_2,pc_2)). \tag{3.9}$$

For a link isolation policy $pc_1 <> pc_2$ which prevents sharing a link in both directions, we add the constraints:

$$\forall n_1.\ \neg(\exists n_2.Fwd(n_1,n_2,pc_1)\ \ \wedge (Fwd(n_1,n_2,pc_2) \vee Fwd(n_2,n_1,pc_2))). \tag{3.10}$$

With single paths for pc_1 and pc_2 (when combined with Equation (3.7)), the above constraints ensure those paths are isolated. Interestingly, for a reachability policy without waypoints, the constraints in Equation (3.7) are not required to enforce isolation. Even though the solver could produce multiple forwarding rules which induce multiple paths, the constraints in Equation (3.9) or Equation (3.10) guarantee isolation as the solver would discard the rules conflicting with another packet class.

3.4 Synthesis of Operator Policies

We now describe how to extend Genesis's synthesis to support various operator policies. Genesis uses hard capacity constraints and optimization objectives pertaining to traffic engineering using linear rational arithmetic (LRA) and linear optimization objectives in SMT. We conclude by describing how Genesis can be extended to allow operators to handle datacenter network failures in a graceful policy-compliant manner.

3.4.1 Link and Switch Table Capacity

For a link capacity policy on the link $sw_1 \rightarrow sw_2 : \omega$, Genesis must ensure that the sum of traffic rates of packet classes using link $sw_1 \rightarrow sw_2$ does not exceed ω. As input, we have the traffic rates $\sigma(pc)$ of each of the packet classes. The constraints added to Ψ are:

$$\sum_{\forall pc} \texttt{ite}(Fwd(sw_1,sw_2,pc),\sigma(pc),0) \leq \omega. \tag{3.11}$$

If a class pc uses link $sw_1 \rightarrow sw_2$, then $(sw_1,sw_2,pc) \in Fwd$ and $\sigma(pc)$ is added in the utilization of the link.

A switch table policy $sw : \gamma$ specifies that the number of forwarding rules on sw must not exceed γ. Similar to the link capacity policy, the constraints ensure the count of all packet

classes which traverse *sw* (each will require a forwarding rule) is $\leq \gamma$:

$$\sum_{\forall pc} \texttt{ite}(\ \exists k.Reach(sw, pc, k), 1, 0) \leq \gamma. \tag{3.12}$$

3.4.2 Traffic Engineering

While the above capacity policies can be used to perform a strict form of traffic engineering (TE) in terms of adhering to link bandwidths, it is often more useful to balance traffic across links because a link failure will affect fewer flows when the flows are spread evenly across the network. To this end, network operators often impose traffic engineering objectives such as minimizing the total link utilization or the maximum link utilization. Genesis performs coarse-grained TE, e.g., given information about diurnal traffic patterns and expected load (such as background systems workloads), plan how to route flows according to a global objective.

Min-tot TE. To perform traffic engineering, link capacities of the network $C(sw_1, sw_2)$ and traffic rates of the packet classes $\sigma(pc)$ are specified as input to Genesis (we assume a single path for a packet class). The utilization of a link $U(sw_1, sw_2)$ is defined as the ratio of total traffic flowing through the link to the link capacity, and encoded using the theory of linear rational arithmetic as:

$$U(sw_1, sw_2) = \frac{\sum_{\forall pc} \texttt{ite}(Fwd(sw_1, sw_2, pc), \sigma(pc), 0)}{C(sw_1, sw_2)} \tag{3.13}$$

The following objective minimizes the total link utilization:

$$\texttt{minimize} \sum_{\forall sw_1, sw_2} U(sw_1, sw_2) \tag{3.14}$$

Min-max TE. To encode the TE objective of minimizing the maximum link utilization, we define a variable *maxU* which represents the maximum link utilization. The constraints added to ensure that *maxU* is greater than or equal to all individual link utilizations:

$$\forall sw_1, sw_2.\ \ maxU \geq U(sw_1, sw_2) \tag{3.15}$$

We then impose the following objective:

$$\texttt{minimize}\ maxU \tag{3.16}$$

Multipath TE. Genesis can support multipath-TE: for a packet class *pc*, we can create *k* subclasses and split traffic of *pc* among the *k* paths adhering to the global objective. However, this requires a static split of traffic among the subclasses. By adding isolation policies to the *k* subclasses, we can split traffic of *pc* along edge-disjoint paths. Furthermore, Genesis can be used for other quantitative objectives like minimizing total latency and load balancing traffic across middleboxes.

3.5 Handling Failures Gracefully

Another network management consideration for operators is the occurrence of failures (switches, links etc.), which are all too frequent in datacenter networks [43]. Failures require recomputation of paths compliant to the input policies for the modified topology. A naive approach is to use Genesis to resynthesize the modified instance; however, the new solution may be drastically different from the original data plane, incurring a large overhead of removing old rules and installing new ones [47, 51].

In what follows, we describe two techniques to handle failures more gracefully. The first technique is data-plane resiliency (§3.5.1), which synthesizes and pre-installs resilient data planes, which even in the event of a bounded number of link failures, continue to satisfy input policies. This technique eliminates the need to resynthesize the forwarding rules for every failure event, but it requires extra backup rules on switches, and cannot capture global operator policies.

Thus, we propose a second mechanism called *minimal repair* (§3.5.2), which can transition from the disrupted data plane to a new policy-compliant one with minimal overhead by minimizing the number of switches whose rule tables are modified. Repair does not incur the extra rule cost of the first approach and can capture all Genesis policies. It is also useful for accommodating incremental policy changes, which occur frequently in cloud datacenters [41]. The main drawback is that it still requires removal/installation of rules when a failure occurs, which can end up being expensive depending on the number of switches involved.

3.5.1 Dataplane Resiliency

In this subsection, we describe the transformation of input policies to provide dataplane *t-resilience* [84], i.e., in the event of up to *t* arbitrary link failures, the synthesized data plane still has a path for each packet class satisfying all policies which is achieved by synthesizing backup paths that satisfy input policies. This approach differs from randomized routing

algorithms which provide resiliency [22] but do not take into account policy-compliance of the backup paths.

We only consider reachability, waypoint, and isolation policies in the input. Global policies like capacity policies and traffic engineering pose a difficulty in synthesis. For example, consider a packet class pc with a traffic rate of $\sigma(pc)$. By considering the backup paths with the same traffic characteristics for synthesis, the total traffic accounted for pc would be $c \times \sigma(pc)$ (for some constant c), leading to under-provisioning of resources. Our current resilience transformation has no provisions to avoid or minimize the under-provisioning of resources which affect capacity policies and TE objectives.

Given the physical topology $T = (S, L)$, we define a link-failure scenario θ as the set of failed links such that $\theta \subseteq L$. We define $\Theta(t)$ as the set of all failure scenarios where no more than t arbitrary links fail,—i.e. $\Theta(t) = \{\ \theta\ \mid\ |\theta| \leq t\}$. Given a packet class pc, we construct the induced data plane graph $\xi = (S, L_{pc})$ from the links of the paths returned by the synthesis algorithm for class pc. For a failure scenario θ, the active data plane $\xi_\theta = (S, L_{pc} \setminus \theta)$ represents all the links used by ξ which are unaffected by the failure scenario. A data plane ξ is *resilient* to θ if it contains a path from the source to destination for the packet class in the active data plane ξ_θ.

Definition 3 (Resilience)**.** *A data plane $\xi = (S, L_{pc})$ for class pc is t-resilient if ξ is resilient to all $\theta \in \Theta(t)$.*

While resilience deals with existence of paths during failure scenarios, we extend the notion to include policy compliance.

Definition 4 (Policy-compliance)**.** *A t-resilient data plane $\xi = (S, L_{pc})$ for class pc is policy-compliant if under any failure scenario $\theta \in \Theta(t)$, any path for pc in $\xi_\theta = (S, L_{pc} \setminus \theta)$ satisfies the input policies.*

Algorithm 1 shows how Genesis can be used to provide *t-resilience*. The idea is to modify the input policies such that multiple disjoint paths satisfying the original policies are synthesized for each packet class. For *t-resilience*, a packet class pc needs at least $t+1$ edge-disjoint paths from source to destination. We ensure this property holds by creating $t+1$ new packet classes ($\hat{pc}$ in line 8) and use *link-isolation policies* amongst all pairs in $\hat{pc}$ (line 10) to create $t+1$ edge-disjoint paths for pc. The synthesized data plane $\hat{\xi} = (S, L_{pc})$ for class pc is constructed from the paths in the resilient packet class set $\hat{pc} = \{rc_1, \ldots, rc_{t+1}\}$, i.e., $L_{pc} = \bigcup_{rc \in \hat{pc}} L_{rc}$. Each path of $\hat{pc}$ satisfies the reachability policy, and any t link failure scenario cannot affect all $t+1$ paths.

However, the resilient paths need to satisfy the input isolation policies with other packet classes (which themselves have $t+1$ paths for resilience). Thus, for a given policy $pc_1 || \ pc_2$,

Algorithm 1 Resilience Transformation

1: [Input] PC: Packet classes (Reachability/Waypoint policies)
2: [Input] I: Isolation policies (Traffic and Link types)
3: [Input] t: Maximum number of arbitrary link failures
4: [Output] PC^R, I^R: Transformed set of policies such that the synthesized data plane is *t-resilient* and policy-compliant

5: $PC^R, I^R \leftarrow \emptyset$
6: **for** $pc : \{src_{pc}, dst_{pc}, W_{pc}\} \in PC$ **do**
7: // Create $t+1$ edge-disjoint paths of pc
8: $\hat{pc} = \{rc_1, \ldots, rc_{t+1}\}$ s.t $\forall m.\ rc_m : \{src_{pc}, dst_{pc}, W_{pc}\}$
9: $PC^R = PC^R \cup \hat{pc}$
10: $I_{pc} = \{rc_m <> rc_n \mid \forall m,n \leq t+1 \wedge m < n\}$
11: $I^R = I^R \cup I_{pc}$
12: **for** $i : pc_m < op > pc_n \in I$ **do**
13: $\hat{i} = \{rc_1 < op > rc_2 \mid \forall rc_1 \in \hat{pc_m}, \forall rc_2 \in \hat{pc_n}\}$
14: $I^R = I^R \cup \hat{i}$
15:
16: **return** PC^R, I^R

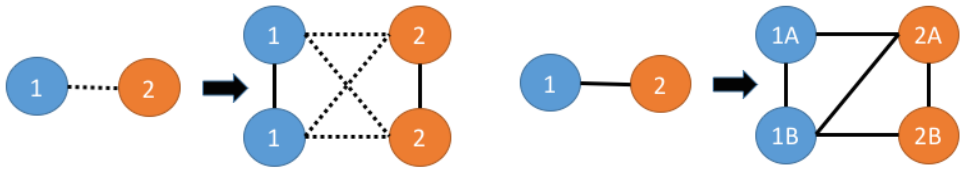
we add isolation policies to every pair of classes of $\hat{pc_1}$ and $\hat{pc_2}$ (line 13). This ensures that any path chosen in the data planes of pc_1 and pc_2 will be isolated from one another, thus providing policy-compliance under any t link failure scenario. Figure 3.3(a) demonstrates an example transformation for providing 1-resilience.
We now that Algorithm 1 is sound.

Theorem 3.5.1 (Soundness)**.** *Given input policies* (PC, I)*, the data plane* $\hat{\xi}_{pc}$ *for every packet class* $pc \in PC$ *synthesized from transformed policies* (PC^R, I^R) *is* t-resilient *and policy-compliant.*

If there are no isolation policies in the input, the resilience transformation in lines 8-11 of Algorithm 1 is complete.

Theorem 3.5.2 (Completeness)**.** *Given input policies* (PC, I) *such that* $I = \emptyset$*, the synthesized data plane* ξ *for a packet class pc is* t-resilient *if and only if it contains* $t+1$ *edge-disjoint paths from source to destination for pc.*

When the original policies contain link-isolation policies, the policies from Algorithm 1 may return `unsat` even when a resilient data plane exists. Specifically, line 13 can add more policies than required for resilience. Figure 3.3(b) shows a transformation required for 1-resilience with a smaller number of link-isolation policies among different classes of pc_1 and pc_2 than one obtained from Algorithm 1. Consider a failure scenario which affects the path of pc_{1A}. By virtue of the link-isolation policies, pc_{1B} and pc_{2A} will be unaffected and can be used as paths for pc_1 and pc_2 respectively, and $pc_1 <> pc_2$ holds. Now suppose pc_{1B}



(a) Traffic Isolation (b) Link Isolation

Fig. 3.3 (a) Resilience Transformation for $pc_1 || \ pc_2$ for providing 1-resilience. The dotted lines represent traffic isolation policies, while the solid lines represent link isolation. (b) Example of a sufficient transformation for 1-resilience in the case of a link-isolation policy.

is affected. Similarly, pc_{1A} and pc_{2A} can be used as the paths for the original packet classes. The same scenarios hold symmetrically for pc_2, and thus the resilience transformation can be achieved without adding link-isolation policies amongst all the packet classes.

3.5.2 Minimal Repair

Dataplane resiliency imposes high rule storage overhead on switches, and cannot accommodate global policies like link capacity bounds. As an alternative to it, we extend Genesis's synthesis algorithm to perform minimal network repair using MaxSMT.

Formally, the MaxSMT problem is as follows: given a set of formulas $\Psi_0, \Psi_1, \ldots, \Psi_n$ with associated weights $w_1, \ldots, w_n$, find a subset $M \subseteq \{1, \ldots, n\}$ s.t: 1) $\Psi_0 \wedge \bigwedge_{i \in M} \Psi_i$ is satisfiable, and 2) The *reward* $\sum_{i \in M} w_i$ is maximized. The constraints $\Psi_1, \ldots, \Psi_n$ denote *soft* constraints, and the associated weights w_i encode the award for including Ψ_i in the satisfying assignment.

We reduce the network repair problem to a MaxSMT problem and use soft constraints to minimize the number of switches on which rules need to be updated. Note that the disadvantage w.r.t. dataplane resiliency is that switches still require rule updates, which may take time depending on the number of switches involved.

Let the policy constraints generated by Genesis for the new network state be Ψ_0, and let $\overline{Fwd}$ be the present data plane that does not satisfy Ψ_0. The objective is to find new Fwd which satisfies Ψ_0 while maximizing the number of *preserved switches* (switches whose rules are unchanged). If the rules on switch sw_i are preserved, then Fwd and $\overline{Fwd}$ have the same forwarding rules for all packet classes which traverse through sw_i. The following soft constraints capture this idea:

$$\Psi_{sw_i} = \bigvee_{\substack{\forall sw_j, pc \\ (sw_i, sw_j, pc) \in \overline{Fwd}}} Fwd(sw_i, sw_j, pc) \qquad w_{sw_i} = 1 \tag{3.17}$$

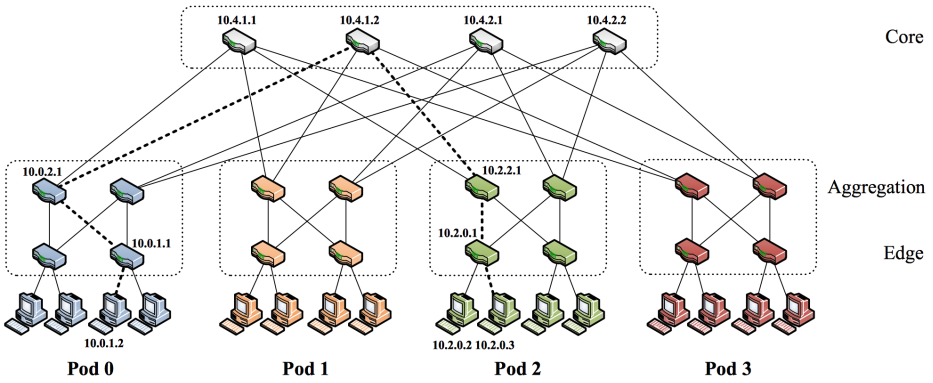


Fig. 3.4 Fat Tree Topology with three-level hierarchy.

The solution to this MaxSMT problem is a data plane that minimizes the number of switches whose rules have to be changed. Alternate repair objectives like minimizing the number of changed forwarding rules can be expressed similarly. Interestingly, the Genesis's network repair mechanism can also be used to transform an existing non-compliant data plane to a policy-compliant one.

3.6 Tactics

Synthesizing a data plane translates to choosing paths from the solution space of all paths for each reachability policy such that the chosen paths satisfy all policies, e.g., waypoint traversal and isolation. Datacenter topologies, e.g., fat-trees [8], have numerous paths between edge switches to provide full bisection bandwidth. Thus, the solution space of paths for a pair of endpoints is large. For example, consider the fat-tree topology in Figure 3.4. The number of paths under length 10 between two edge switches in the same pod is 242 and between two edge switches in different pods is 272. If we consider the synthesis of n packet classes, the problem roughly translates to finding a solution in the space of size $(242)^n$. Operators can leverage the network structure of topologies to reduce the solution space by specifying undesirable path patterns. For example, the operator might require that a path between two edge switches in a fat-tree does not traverse another edge switch. This pattern doesn't drastically reduce the set of possible paths due to the dense interconnect between aggregate and core switches.

We introduce *tactics* (the name is inspired from the usage in SMT solvers, not proof assistants) on labels; abstractions that allow a network operator to impose restrictions on paths. We map the set of switches to labels to have a coarse-grained way for specifying path patterns. Tactics on labels help create search strategies which can be used for groups of packet classes instead of individual switch-level patterns which lack generality.

Let Lb be the set of labels and S be the set of switches in the topology. Let $\phi : S \to Lb$ be the labeling function that maps each switch to a label in Lb. E.g., we can leverage the hierarchical structure of the fat-tree by mapping all switches in the same level (core, aggregate or edge) to the same label. A path p is a word over the alphabet S. We define the path-labeling function $\Phi : S^* \to Lb^*$, which maps each switch in the path to its corresponding label. For example, given the path $p = e1\ a2\ c3\ a4\ e2$, the path-labeling function produces $\Phi(p) = eacae$—the function maps each switch to its corresponding label. Here, e, a, and c stand for edge, aggregate, and core respectively.

Tactics are simple regular expressions over the set of labels and are used to blacklist certain path patterns. Regular expressions have been previously used in tools like NetGen [77] to specify the paths for a packet class. While supporting full regular expressions is possible, it causes a blow-up in the solving time as further constraints need to be added to the solver to ensure that a path satisfies the regular expression. Rather than specifying how the path must look like, we use regular expressions on switch labels to specify *blacklists* i.e., what the path must *not* look like. A tactic, for example, can blacklist paths from an edge switch to an edge switch that go through another edge switch.

3.6.1 Restricted Tactic Syntax

We specify tactics using a restricted set of regular expressions[4] that not only do not require extra constraints to be added, but actually allow us to reduce the number of constraints in Ψ. Tactics are regular expressions described by the following grammar:

$$\begin{aligned} R &:= \neg(l_{src}.^{i}C.^{*}l_{dst}) \\ C &:= \varepsilon \mid l_1 \mid l_1 l_2 \end{aligned}$$

where $l_i \in Lb$ and l_{src}, l_{dst} are used to specify the labels of the source switch and destination switch, respectively. Since our goal is to blacklist paths, we allow regular expressions to be negated at the outer level.

Example 1. *$\neg(e.^{i}c.^{*}e)$ indicates that the path must not contain a core switch at the $(i+1)^{th}$ step. Similarly, $\neg(e.^{i}.^{*}e)$ indicates that the path connecting two edge switches should have a length $< i+1$.*

Let $\pi = sw_0 \ldots sw_k$ be a path for packet class pc and let its labeling be $\Phi(\pi) = a_0 \ldots a_k$. We say that π satisfies a tactic R, $\Phi(\pi) \in L(R)$, if the following holds:

- $\Phi(\pi) \in L(\neg R)$ iff $\Phi(\pi) \notin L(R)$;

[4] It is a subset of star-free languages [28].

- $\Phi(\pi) \in L(l_{src}.^{i}.^{*}l_{dst})$ iff $k \geq i+1$, $l_{src} = a_0$, and $l_{dst} = a_k$;
- $\Phi(\pi) \in L(l_{src}.^{i}l.^{*}l_{dst})$ iff $k \geq i+2$, $l_{src} = a_0$, $l_{dst} = a_k$, and $a_{i+1} = l$;
- $\Phi(\pi) \in L(l_{src}.^{i}l_1l_2.^{*}l_{dst})$ iff $k \geq i+3$, $l_{src} = a_0$, $l_{dst} = a_k$, $a_{i+1} = l_1$, and $a_{i+2} = l_2$.

In Genesis, operators can specify conjunctions of tactics which adhere to the restricted syntax and the synthesis algorithm is modified to enforce the tactics.

Example 2. *The "No Edge" tactic ensures that a edge-edge path cannot traverse another edge switch. It is expressed using conjunctions of tactics:* $\neg(e.^{*}e.^{*}e) \equiv \bigwedge_{i=0}^{i=\mu-2} \neg(e.^{i}e.^{*}e)$ *where* μ *is the limit on path length. The "Valley-free" Tactic:* $\neg(e.^{5}.^{*}e) \wedge \neg(e.^{*}e.^{*}e)$ *ensures that a edge-edge path is of the form* $e-a-c-a-e$.

Paths used in production datacenter networks adhere to both these tactics [45, 79]. Such paths are simple and make networks easy to manage and troubleshoot [14].

3.6.2 Modified Synthesis Algorithm with Tactics

In our synthesis algorithm, the reachability-propagation constraints (Equation (3.3)) construct the path from destination to source. We use tactics to prune these constraints, so that the path synthesized satisfies the tactic regular expression.

The tactic set $\Gamma = \{(R_1, pc_1), \ldots, (R_n, pc_n)\}$ specifies that tactic R_i is applied on packet class pc_i where $pc_1, \ldots, pc_n \in PC$ and $R_1, \ldots, R_n$ are regular expressions satisfying the restricted tactic syntax. Given a tactic R applied to a packet class pc, we define $\Psi_T(R, pc)$ as the additional SMT constraints used for synthesis such that $(\Psi \wedge \bigwedge_{(R,pc)\in\Gamma} \Psi_T(R, pc))$ is provided as input to the SMT solver. Note that $\Psi_T(R, pc)$ is presented as additional SMT constraints only for clarity. In practice, the modified synthesis algorithm will remove constraints for each $(R, pc) \in \Gamma$.

Type 1. For a tactic R of the form $\neg(l_{src}.^{i}.^{*}l_{dst})$ applied to pc:

$$\Psi_T(R, pc) = \forall sw, k \geq i+1.(sw, pc, k) \notin Reach \tag{3.18}$$

This tactic restricts the path to a length $< i+1$. Thus, we can remove the reachability constraints of Equation (3.3) for all the tuples $(sw, pc, k) \notin Reach$ satisfying Equation (3.18) as they cannot contribute to any path satisfying the tactic.

Type 2. For a tactic R of the form $\neg(l_{src}.^{i}l.^{*}l_{dst})$ applied to pc:

$$\Psi_T(R, pc) = \forall sw.\ \phi(sw) = l \wedge sw \neq dst \implies (sw, pc, i+1) \notin Reach \tag{3.19}$$

The tactic ensures that a switch with label l cannot be reached in $i+1$ steps, except if $l = l_{dst}$. In that case, only the destination switch with label l can be reached in $i+1$ steps as the path with labeling $l_{src}.^{i}l_{dst}$ satisfies the tactic. If $l \neq l_{dst}$, then all switches with label l cannot be reached in $i+1$ steps. For all tuples $(sw, pc, i+1) \notin Reach$ satisfying Equation (3.19), we can remove the reachability constraints of Equation (3.3).

Type 3. For a tactic R of the form $\neg(l_{src}.^{i}l_1 l_2.^{*}l_{dst})$ applied to pc:

$$\Psi_T(R, pc) = \forall n_1, n_2.\ \phi(n_1) = l_1 \ \wedge\ \phi(n_2) = l_2 \ \wedge\ n_2 \neq dst \implies \neg(Reach(n_1, pc, i+1) \wedge Fwd(n_1, n_2, pc)) \quad (3.20)$$

This tactic ensures that a switch with label l_1 at $i+1$ in the path will not forward the packet to a switch with label l_2 (unless n_2 is the destination). To enforce this, we modify the Equation (3.3) and remove all $l_1 \rightarrow l_2$ edges at position $i+1$ in the path for which the switch with label l_2 is not the destination.

We now state the soundness and completeness of the synthesis algorithm with tactics. Let $(Fwd, Reach)$ be a model of Ψ and $\Pi = \texttt{paths}\ (Fwd,\ Reach)$ be the set of induced paths (from Definition 1).

Theorem 3.6.1 (Soundness)**.** *For a tactic set* Γ*, if* $(Fwd, Reach) \models \Psi \wedge \bigwedge_{(R,pc)\in\Gamma} \Psi_T(R, pc))$*, then* $\forall (R, pc) \in \Gamma.\ \forall (\pi', pc') \in \Pi.\ pc = pc' \implies \Phi(\pi') \in L(R)$.

Theorem 3.6.2 (Completeness)**.** *For a tactic set* Γ*, if* $\Pi \models \Psi$ *and* $\forall (R, pc) \in \Gamma.\ \forall (\pi', pc') \in \Pi.\ pc = pc' \implies \Phi(\pi') \in L(R)$ *then* $\forall (R, pc) \in \Gamma.(Fwd, Reach) \models \Psi_T(R, pc)$.

The intuition behind the restricted tactic syntax comes from the structure of the reachability propagation constraints (Equation (3.3)) which construct the path for a packet class. Each constraint enforces that if a switch is reachable in k steps in a path, there must a neighbour switch in the path reachable at $k-1$ steps. Using the *Reach* relations, we can specify path length restrictions (Type 1) or prevent switches with a certain label at some position (Type 2). The structure of Equation (3.3) restricts regular expressions on only local switch neighbours (Type 3). The structure of these constraints prevents us from being able to specify unrestricted regular expressions (supporting these would require adding additional constraints).

Example 3. *Consider a tactic* $\neg(e.^{i}aca.*e)$*. To enforce this tactic, we need to have constraints which prevent the path reaching an aggregate switch in* $i+3$ *steps when the path traverses an aggregate and core switch at* $i+1$ *and* $i+2$ *steps respectively. This cannot be specified by modifying the reachability constraints in its current form, because the constraints for reachability for a switch in* $i+3$ *steps only depends on the constraints for reachability in* $i+2$ *steps.*

Tactics are heuristics, but well-defined ones with formal semantics and provable soundness properties. In practice, tactics can be used to specify restrictions which would not reduce the search space dramatically, but are still useful toward speeding up the synthesis, especially in datacenter topologies which are hierarchical and can be used to specify interesting tactics. One of the biggest advantages of tactics is that tactics are *policy-agnostic* since they enforce conditions on the path, and can be used in conjunction with the different policies supported by Genesis (isolation, traffic engineering etc.). Thus, we have provided a framework for the development of search strategies based on path properties, and operators can design tactics based on the physical topologies (datacenter topologies are hierarchical) to create a library of tactics that can be reused for workloads. While tactics sacrifice completeness, operators can discard the tactic if synthesis fails and use Genesis without tactics.

3.7 Divide-and-Conquer Synthesis

Since the complexity of finding a data plane enforcing policies is *exponential* in the number of packet classes, the synthesis time shoots up with increasing packet classes. However, since datacenter topologies have a dense interconnection of links between layers there can be numerous different data planes as solutions. We propose to speed up synthesis by partitioning the problem into smaller components.

Suppose we have two packet classes pc_1, pc_2 isolated from one another; the standard synthesis algorithm adds constraints for both packet classes to Ψ and finds the solution for both classes. Instead, we could synthesize pc_1 independently and, after that, find a solution for pc_2 that is isolated from the path obtained for pc_1. However, synthesis of pc_2 can fail because the solution of pc_1 may be such that there is no way to place an isolated path for pc_2 but if they had been synthesized together, a solution might exist. To tackle this, we use a recovery mechanism which uses unsatisfiable cores extracted from the solver. We term this procedure *divide-and-conquer* synthesis.

We define a policy graph $P = (R, I)$ where every vertex $r \in R$ is a packet class for a reachability/waypoint policy and edges denote isolation constraints between packet classes. An edge $(r_1, r_2) \in I$ means that the paths of $r1$ and $r2$ are isolated from each other. We assume that there are no capacity policies in the input specifications. Given the policy graph P, we can synthesize each connected component independently, since packet classes in different connected components are not constrained by an isolation policy, and therefore are independent of each other.

We describe the divide-and-conquer synthesis procedure in Pseudocode 2, which takes as input a connected component of the policy graph. The crux of the procedure is that we

Pseudocode 2 Divide-and-Conquer Synthesis

```
1:  procedure DCSYN(P)
2:      if size(P) < P_thres then
3:          Apply normal synthesis on P
4:      else
5:          Partition P into P_1 and P_2
6:          if interpartition edges > E_thres then
7:              Apply normal synthesis on P
8:          F = []          /* Failed solutions */
9:          attempts = 0
10:         while attempts < RA_max do
11:             sol_1 = Apply synthesis on P_1 such that sol_1 ∉ F
12:             if synthesis(P_1) fails then
13:
14:                 return DCSyn failure
15:             sol_2 = Apply synthesis on P_2 such that
                  sol_1 ∪ sol_2 is a solution for P
16:             if synthesis(P_2) fails then
17:                 F.append(unsat-cores(P_2))
18:                 attempts++
19:             else
20:
21:                 return DCSyn success
22:
23:         return DCSyn failure
```

partition each connected component P into two smaller components P_1 and P_2 and do the following: 1) synthesize P_1 and obtain a solution S_1, and 2) for packet classes of P_2 that are isolated to packet classes in P_1, add the solution S_1 as a constraint to ensure that the packet classes in P_2 will not share the edges of the respective paths obtained in P_1.

We use the *min-cut* to partition the policy graph connected component into two such that the number of edges between the partitions is minimized. The rationale is that since we intend to perform the synthesis of both partitions separately, the partitioning should maximize isolation policies within components and minimize those across components. By maximizing isolation policies during synthesis of the component, the partial solution is more likely to be compatible with a complete solution. However, if the min-cut partitioning produces a component smaller than a threshold size, we perform partitioning of the graph into two *equal sized* partitions and minimize the cut edges between the partitions. We need to ensure that we don't partition the graph smaller than a threshold, as the partial solutions obtained

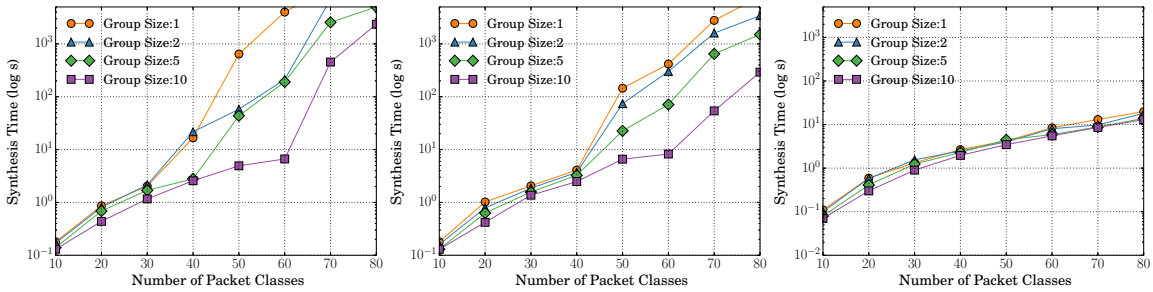
by synthesis of very small partitions are more likely to conflict with other packet classes. Genesis performs divide-and-conquer synthesis recursively on the components till we cannot partition the component further.

Solution Recovery. While in the best case divide-and-conquer synthesis leads to a great increase in performance, we need a recovery mechanism in case we cannot find compatible partial solutions. Many SMT solvers track constraints and return an unsatisfiable core [24] when synthesis fails. Informally, the unsatisfiable core is a set of tracked constraints that describes why there wasn't a feasible solution. This helps us track failed partial solutions. Thus, if synthesis of P_2 fails, the unsatisfiable cores describe what paths of the solution of P_1 are causing the synthesis of P_2 to fail. When performing synthesis of P_1 again, we therefore ensure that we get *different* paths from those extracted from the unsatisfiable cores. Basically, we perform a *solver-guided* enumeration of different solutions of P_1 to find a satisfying solution for P_2. The solution recovery procedure is described in lines 10- 21.

Since, recovery is a form of enumeration, in cases where the graph has greater number of policies (for e.g., a clique), finding a solution could take a large number of enumerations, while synthesis without partitioning would provide a solution faster. Thus, we bound the number of enumerations performed by the recovery mechanism and return failure if we don't obtain a solution.

Divide-and-conquer synthesis with recovery is sound, but it is incomplete as we bound the number of enumerations. The success of this approach is directly related to the size of the components (determined by P_{thres}). This is because, by synthesizing more packet classes together, we decrease the conflicts arising between partial solutions. The extreme case of when we do not partition the component at all (normal synthesis) is complete. To make the synthesis complete with faster convergence, we perform iterations of divide-and-conquer synthesis, and at each iteration we double the partition threshold P_{thres} if the previous iteration failed. This scheme tries to balance the trade-off between completeness, which requires larger components, and performance, as synthesis is faster on smaller components. In the extreme case, after $O(log\ P)$ iterations, $P_{thres} > P$ and divide-and-conquer will not partition P and yield the solution (if one exists).

Divide-and-conquer is more *effective* when there is a large number of solutions and partial solutions do not fail. When the problem is highly constrained and the number of solutions is low, the recovery mechanisms and multiple iterations could lead to a degraded performance. A drawback of the divide-and-conquer approach is that it is difficult to apply to global policies (like traffic engineering) primarily because splitting the input problem isn't easy; development of strategies to speed-up global policies is future work.



(a) Baseline (b) No Edge Tactic (c) Valley-free Routing Tactic

Fig. 3.5 Total synthesis time (log scale) for isolation workloads over range of packet classes and different tenant-group sizes.

3.8 Evaluation

We implemented a full working prototype of Genesis in Python. We have implemented an interpreter for the Genesis Programming Language using PLY [4] and the synthesizer using the SMT solver Z3 [27] and its νZ extension for MaxSMT and linear optimization [18]; Genesis outputs the forwarding rules for the switches, which can be provided as input to a SDN controller (e.g., Floodlight [2]) to install over the network. Genesis uses the Metis graph-partitioning library [53] to perform equi-sized partitioning used by divide-and-conquer synthesis.

In this section, we evaluate Genesis using enterprise-scale multi-tenant data center settings. Specifically, we ask:

- What is the performance of Genesis's baseline synthesis algorithm for tenant policies? How does the performance vary with size of the network, number and the nature of policies in use? (§3.8.1)
- What is the performance of Genesis for operator policies like capacity bounds, traffic engineering, and network repair which use SMT with linear optimization objectives and MaxSMT? (§3.8.2)
- How much do tactics help improve Genesis's performance? Which tactics offer the best improvement? (§3.8.3)
- To what extent does the divide-and-conquer synthesis improve Genesis's performance? When does it lead to degraded synthesis times? (§3.8.4)

Our experiment settings have a few thousand servers, tens of switches, and hierarchical fat-tree network topologies which reflect a private datacenter. Our experiments are parameterized by: (a) total size of the fat-tree network (45-180 switches), (b) number of tenants (1-80), and (c) number of packet classes in a tenant (1-10). Note that a single packet class can be used to

specify policy for multiple host-pairs of a tenant connected to the same edge switches, and placement of the hosts can take uniformity of policy in account to reduce the explosion of packet classes with increasing hosts.

Our primary metric of interest is synthesis time, measured in seconds. In measuring this, we focus on the time the Z3 solver takes to solve the constraints.[5] All experiments were conducted using a 32-core Intel-Xeon 2.40GHz CPU machine and 128GB of RAM. For evaluating the baseline performance, we impose a synthetic limit on the path length μ to be 10, which is adequate for a fat-tree topology with three levels.

3.8.1 Baseline Synthesis Performance for Tenant Policies

Multi-Tenant Isolation. To evaluate the baseline performance of Genesis, we model a multi-tenant 80 switch topology with tenant-isolation in Figure 3.5(a). For each workload we have n tenants with group size g which is the number of packet classes for each tenant. The x-axis shows the total packet classes $n * g$. Packet classes of a tenant are not isolated (and they implement simple reachability within the tenant), while packet classes of different tenants are traffic-isolated. Thus, no two tenants share a link in the same direction, and can never affect each other's performance. We randomly[6] place endpoints for the tenants' packet classes, ensuring that no more than 4 tenants share a single edge switch. Operators can aggregate a tenant's traffic from multiple instances connected to the same switches as a single reachability policy and establish pathways for communication amongst different switches.

For a fixed group size, we observe that the total synthesis time increases exponentially with number of packet classes. As group size decreases, for the same number of classes, the number of tenants increases, increasing the number of isolation policies and the synthesis times. Group size 1 denotes the extreme case where all flows are isolated with each other.

While we evaluated a multi-tenant isolation setting, there are other scenarios that translate to these workloads. Consider an example where specific flows of tenants require QoS guarantees and these flows must be isolated w.r.t. all other flows. This translates to a two-tenant isolation setting. Operators can provide weaker isolation such that two flows must be isolated on only certain "special" links. This is an easier problem to tackle than isolation over all links, and the performance of such scenarios would be better. Failure resiliency

[5] We do not account for constraint generation time in our evaluation, as it has polynomial time complexity and thus, can scale well unlike constraint solving time; a well-engineered system can considerably reduce the constraint generation overheads.

[6] Smarter placement of tenants could speed-up synthesis as tenant endpoints would be located closer to each other. The placement algorithm can be used to develop specialized tactics.

uses link-isolation policies which exhibit a similar performance compared to the workloads considered here.

Effect of Topology Size. To evaluate Genesis across increasing topology sizes for isolation workloads, we fix the tenant-group size to 5, and for each topology, we maintain the ratio of packet classes to number of edge-aggregate links to 0.25. We choose this metric because if we keep the number of classes constant, as topology sizes increases, it is easier to find isolated paths due to more links. Thus, by keeping the number of packet classes proportional to size of the topology, we maintain the relative difficulty of the workload across topologies. We show the average synthesis time per class with increasing topology sizes in Figure 3.7 (baseline trace). We are able to synthesize forwarding rules for 12 tenants with group size 5 in a 125 switch topology in 124 seconds (avg. 2 seconds per traffic class). We also observe that average time per flow increases exponentially with larger topologies, thus synthesis times are also exponential in the number of switches.

Waypoint Policies. To evaluate Genesis's performance for ordered sets of waypoints, we fix the number of waypoints (range:1-5) and generate 100 waypoint policies with different sizes and permutations of the ordered waypoint sets for a 80 switch topology. Each policy has edge switches as endpoints and randomly picked core or aggregate switches for waypoints. The synthetic limit μ on the path length is increased to 15 and no tactics are used (difficult to devise a tactic for the path satisfying a waypoint policy). The average synthesis time for a waypoint policy is reported in Table 3.2. We observe that synthesis time increases exponentially with total number of waypoints in a packet class's policy, owing to the complexity of the problem. Genesis can synthesize rules for a path with 3 total waypoints in less than a second, on average.

3.8.2 Baseline Synthesis Performance for Operator Policies

Isolation with Link Capacity Policies. Figure 3.6 (baseline trace) shows the average synthesis time per flow for the same setting as above, but additionally, there are 10 low-bandwidth links in the network for which the operator specifies capacity policies (all packet classes have uniform capacity). Since we use LRA for link capacity constraints, we see an increase in average time for synthesis when compared to pure isolation which is completely encoded using SAT.

Traffic Engineering. Table 3.3 shows the synthesis time for workloads on a 80-node fat-tree topology with different traffic engineering (TE) objectives. Genesis can synthesize a data plane minimizing average utilization for 200 packet classes in approximately 2000 seconds. However, for minimizing the maximum link utilization, Genesis can only synthesize 50 packet classes in close to 4000 seconds. For both objectives, the synthesis time increases

Number of Waypoints	Avg. Synthesis time per Class (s)
1	0.034
2	0.138
3	0.983
4	15.41
5	32.93

Table 3.2 Average synthesis time per class for waypoint policies with increasing number of waypoints.

exponentially with the number of packet classes. SMT with optimization objectives is an emerging field of research, and we envision that solvers in the future will become fast and handle larger workloads.

Minimal Repair. To evaluate the performance of minimal repair using MaxSMT, we consider a setting with 8 tenants, each with 10 packet classes (total classes=80), and tenant flows are isolated from one another. Now, we disable the switch with the largest number of rules, and try to find a new data plane satisfying the original tenant isolation policies such that the number of switches unaffected is maximized. We can synthesize the minimal repair in nearly 200 seconds on average. With repair, the new data plane only changes rules on 2-3 switches on average, while naive synthesis results in nearly 60 switches being updated, which is very expensive.

3.8.3 Tactic Reductions

We demonstrate the improvements from using tactics for isolation workloads with different number of tenants and group sizes on a 80 switch topology.

"No Edge" Tactic: Figure 3.5(b) shows the synthesis time for isolation workloads using the no edge tactic $\neg(e.^*e.^*e)$, which has a best-case speedup of 9.5$\times$ over baseline synthesis. Using this tactic, Genesis can synthesize forwarding rules for 12 tenants with group size 5 in under 200 seconds.

"Valley-free" Tactic: For the same isolation workloads as above, we use the tactic $\neg(e.^5.^*e)$ $\wedge\neg(e.^*e.^*e)$ which ensures *valley-free routing*, that is paths are of the form *eacae*. The results are shown in Figure 3.5(c). Using this tactic, Genesis synthesizes forwarding rules for each workload in under 20 seconds and can achieve a best-case reduction of 400$\times$ compared to synthesis without tactics.

Effect of Topology Size: In Figure 3.7, we evaluate the performance of different tactics for different topology sizes. There is a significant reduction in synthesis time for each tactic when compared to the baseline synthesis. The performance of each tactic is directly related to the reduction of the search space: more restrictive tactics have lower synthesis times.

Workload Type	Description	Time (s)
minimize-avg-te	100 packet classes	425
	200 packet classes	2002
minimize-max-te	25 packet classes	522
	50 packet classes	4192
Network repair	8 tenants, group size 10, tenant-isolation, 1-switch failure	219

Table 3.3 Synthesis times for workloads on a 80-node fat-tree topology with different optimization objectives.

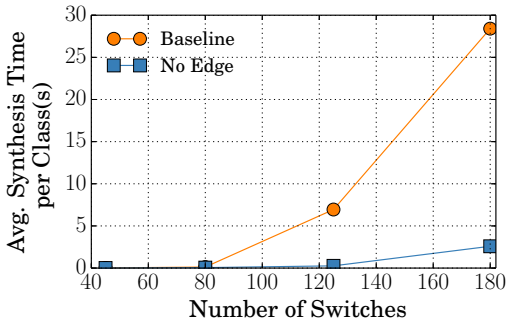


Fig. 3.6 Average synthesis time per packet class versus topology size for isolation workloads with the ratio of packet classes to number of edge-aggregate links 0.25 and 10 low bandwidth links in the topology have capacity policies.

Using the $length \leq 7$ tactic and "no edge" tactic, Genesis synthesizes forwarding rules for 20 tenants of group-size 5 in 100 seconds in a 180 switch topology (9× speedup over synthesis without tactics).

Isolation with Link Capacity Policies: A similar setup with additional link capacity constraints for 10 links is evaluated using the no edge tactic, and we get a best-case 14× improvement over baseline synthesis. Tactics can provide a considerable improvement over the baseline performance as illustrated by these experiments, and demonstrate the viability of synthesis approach of Genesis to real-world networks.

3.8.4 Divide-and-Conquer (DC) Synthesis Performance

To evaluate the divide-and-conquer (DC) synthesis procedure, we perform 100 runs of DC and non-DC synthesis (with the no edge tactic in both cases) on isolation workloads with varying number of tenants and different group sizes used in §3.8.1. We compute the speedup (time of non-DC synthesis/time of DC synthesis) and plot its cumulative frequency distribution in Figure 3.8 to quantify the benefits of DC synthesis. For more than 80% of the workloads, divide-and-conquer offers better or comparable performance to non-DC synthesis,

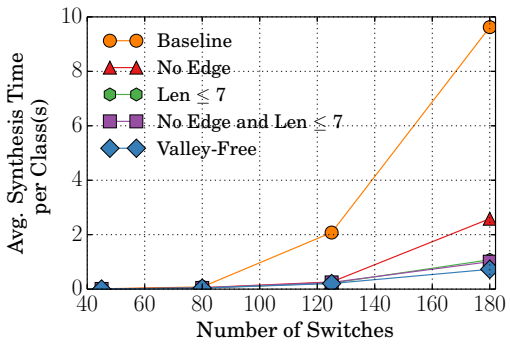


Fig. 3.7 Average synthesis time per packet class versus topology size for isolation workloads w/o different tactics with the ratio of packet classes to number of edge-aggregate links 0.25.

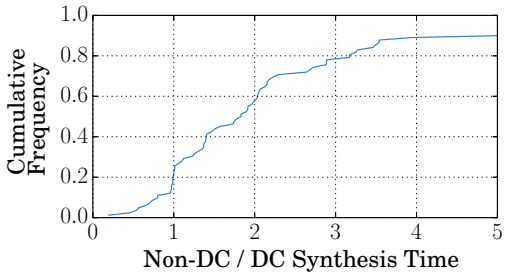


Fig. 3.8 CDF for speedup achieved by divide-and-conquer synthesis.

achieving a speedup of 2× for nearly 40% of the workloads. For 20% of the workloads, divide-and-conquer performs worse than the non-DC approach, especially for workloads with tenant group size 1 due to greater number of recovery attempts.

Summary. The key points of our evaluation are: 1) For a representative tenant-group size of 10 in a 80 switch fat-tree, the baseline synthesis performance for synthesizing the forwarding rules for 1 to 8 tenants with complete tenant-isolation is in 0.1-2000s. 2) Operator policies like optimization objectives for TE and network repair is more expensive than synthesis without objectives. 3) Tactics provide considerable speedup over the baseline synthesis. We can synthesize the above workloads in 0.1-300s using the no edge tactic, and under 12s using the valley-free routing tactic. 4) Genesis can further benefit from divide-and-conquer (DC) synthesis, which provides a 2.0× speed-up over non-DC synthesis in 40% of the workloads, in addition to the tactic improvements.

3.9 Related & Future Work

One Big Switch: Kang et al. [52] tackle a similar problem of flow policy enforcement. However, their end-point policies deal with simple reachability. Their rule placement

algorithm takes the path of the flow in the network (called the routing policy) as an input. Zhang et al. [94] build on the "one big switch" abstraction [52] to optimize for the specific case of distributed firewall policy enforcement using ILP. PGA [70] provides a graph-level abstraction for specifying network policies like ACLs and middlebox service chaining. However, PGA abstracts the underlying network as "one big switch" and cannot be used to compose policies like tenant isolation or traffic engineering.

Controller synthesis: Program synthesis has seen limited applications to SDN controllers [93, 67]. These systems synthesize the behavior of individual switches (e.g., learning switches or firewalls); furthermore, these techniques apply to networks operating in a reactive mode (where the first packet of a connection is processed by the controller to determine the actions to employ). Such switch-centric approaches are too constraining and cannot be applied to realize network-wide objectives considered in Genesis. Synthesis has been also used for generating consistent network updates [62, 96]. But this problem is orthogonal to policy enforcement.

Policy languages: The closest approaches to ours are Merlin [83], NetGen [77] and NetKAT [9]. In Merlin, data planes that adhere to policies expressed using regular expressions are synthesized by first intersecting the topology with the regular expressions appearing in the policies and then encoding reachability in the intersected graph using mixed integer linear programming (ILP). Merlin supports minimum and maximum bandwidth guarantees. In its current iteration, Merlin's encoding does not support isolation policies, but we believe that it could be extended to support them. A more prominent difference arises with unordered waypoint policies: expressing a policy including a waypoint set W of size k requires a regular expression of size exponential in k as all the possible permutations of the elements of W must be considered. This fact clearly impacts the performance of Merlin's compiler that would have to generate a mixed ILP with a large number of variables. In Genesis, this is not the case as waypoints can be encoded with polynomially many constraints. While this does not affect the theoretical complexity, our compiler does not incur an a-priori exponential blow-up; it rather relies on the power of SMT solvers to guide the search. This is one of the main aspects behind our decision of not using regular expressions to express policies. Genesis uses a restricted form of regular expressions as tactics that leverage the network topology. While in Merlin, regular expressions *increase* the number of constraints generated by the compiler, tactics *decrease* the number of generated constraints, therefore speeding up the search. To the best of our knowledge, this is the first use of constraints that leverages the topology structure to simplify the search.

In NetGen, network updates that adhere to policies expressed using regular expressions are synthesized using SMT solvers. Given a specification which mentions the packet classes,

the old path, and the new path, NetGen solves the network change problem using an SMT solver. Due to the use of regular expressions NetGen also suffers the limitations we just discussed for Merlin. Interestingly, NetGen uses a specific encoding of regular expressions based on uninterpreted functions that helps reduce the number of constraints. While this encoding is fast when updating a single path, we do not see a way to extend it to our global synthesis setting. A crucial aspect of NetGen is that in its problem formulation each path can be synthesized independently and without affecting the other already synthesized paths. This is not the case when supporting isolation policies: if an old path needs to be moved to satisfy a new policy (e.g., because a link is under maintenance), re-synthesizing such a path can require re-synthesizing other paths.

NetKAT is a domain-specific language and logic for specifying and verifying network packet-processing functions for SDN, based on Kleene algebra with tests (KAT). Semantically, a NetKAT predicate and policy is a function that takes a packet history and produces a set of (possibly empty) packet histories. NetKAT can be used to express certain network-wide policies like reachability, waypoints using regular expressions for describing the paths, and programs on virtual topologies; it uses BDDs and symbolic automata to translate global programs to local switch programs [81]. However, the NetKAT semantics cannot be used to express policies based on hyperproperties [26], i.e., the packet processing function requires multiple packet histories as input. Traffic engineering or isolated paths are policies based on hyperproperties.

Chapter 4

Zeppelin: Synthesis of Fault-Tolerant Distributed Router Configurations

Our previous work Genesis was intended to enforce high-level intents in a Software-defined network (SDN) setting where network forwarding can be programmed from a centralized controller. However, traditional control planes rely on distributed protocols such as Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP) to compute paths; these protocols typically employ variants of least cost path computation, and react to failures by recomputing least cost paths and installing forwarding state in routers that induces the paths. In contrast to centralized SDN, traditional control planes offer greater fault tolerance; but, determining the appropriate distributed realization of policies is hard [11].

Our high-level goal is to develop a system to *automate the process of creating a correct and failure-resilient distributed realization of policies in a traditional control plane*. To be useful, such a system must satisfy a few key requirements. (1) It must handle a wide range of commonly-used policies—including reachability, service chaining, and traffic engineering—to meet applications' diverse security and compliance requirements. (2) It must ensure that configurations are resilient to network malfunctions such as link failures. (3) It must provide support for realizing hierarchical control planes—where a network is split into several "domains" atop which a hierarchy of intra- and inter-domain control plane configurations is deployed—to ensure scalability for large networks [60]. (4) To improve manageability and network cost-effectiveness [41, 17], it must ensure that configurations obey certain general rules-of-thumb [15], e.g., limiting the number of lines of configurations and the use of certain configuration constructs.

Thus, our work contrasts with prior efforts which suffer from one or more drawbacks: they generate SDN- or BGP-specific control planes for a limited range of policies (e.g., peering) [9, 83, 93, 73, 71, 48, 11]; do not attempt to be resilient to failures [29]; do

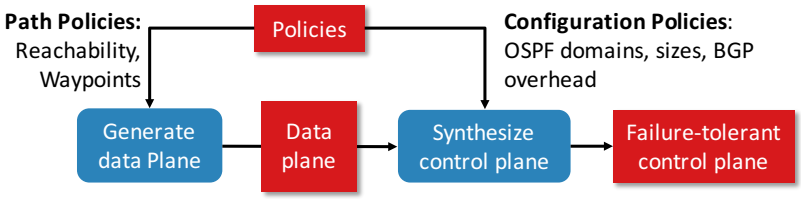


Fig. 4.1 Two-phase process for generating a control plane with failure-tolerance properties

not support generating hierarchical control planes [11]; and do not enable generation of simple-to-manage network configurations.

The problem of synthesizing router configurations for which the distributed control plane is resilient to failures and generates policy-compliant paths is computationally hard. Even generating a set of policy-compliant paths for an SDN is computationally hard—e.g., enforcing isolated paths is NP-complete. While it is possible to develop algorithms for individual policies, accommodating multiple policies and the above requirements is extremely difficult.

An attractive possibility, motivated by recent progress in program synthesis, is to use Satisfiability Modulo Theories (SMT) solvers to automatically search the space of distributed configurations for a suitable "solution"— i.e., concrete router configurations that satisfy input policies and the aforementioned four requirements. SMT solvers provide support for constraint solving for propositional logic (SAT) and linear rational arithmetic (LRA); these powerful theories can encode properties on network paths and configurations. Thus, an SMT-search based approach can, in theory, provide support for multiple policies and different manageability requirements. This approach also decouples the requirements from the underlying search, and thus, could be extended to support new policies.

However, using SMT solvers in our context is non-trivial. To infer the concrete set of paths induced by network configurations, one has to incorporate into synthesis complex concepts—e.g., reasoning about shortest path algorithms requires constraints in complex theories (SAT and LRA). Even with recent advances in SMT solving, approaches that directly generate configurations from policies do not scale to moderately-sized networks or sets of policies [29]. Furthermore, generating resilient control planes needs reasoning about how protocols react to failures, which further complicates an already intractable synthesis problem. Control plane hierarchy and manageability requirements further complicate the problem.

In this work, we present Zeppelin, a system that overcomes the above challenges in SMT-based distributed configuration synthesis. Zeppelin uses a two-phased approach for tractability (Figure 4.1) that does not attempt to generate a policy-compliant control plane in a single step. First, Zeppelin uses Genesis [86] to synthesize paths—i.e., the network forwarding state—that are compliant with given policies, such as, waypoints and isolation.

Zeppelin then generates intra-domain (shortest-path OSPF) and inter-domain (BGP) router configurations that induce the forwarding state synthesized by Genesis *and* provide high resilience. Zeppelin caters for moderate-sized enterprises and multi-tenant datacenters which require support for diverse policies.

We consider three settings with progressively more complex policies and resilience requirements and show how Zeppelin, using its two-phase approach, can effectively generate highly-resilient and policy-compliant solutions.

First, we consider a setting in which the operator of a hierarchically structured network wants to enforce a large diverse set of complex policies (waypoint traversal, isolation, traffic engineering, etc). The operator also desires a simple notion of *connectivity-resilience*—i.e., under most common link failures, even if operator-specified policies (like waypoints) are not enforced, majority of packets can still reach their destination. In this case, Zeppelin synthesizes OSPF configurations using linear constraint solving to compute link weights, and uses the unsatisfiable cores of failed solving attempts to judiciously place a small number of static routes. Zeppelin synthesizes BGP configurations directly from the domain mapping (i.e., which router belongs to what domain) and the paths. Using these techniques, Zeppelin can generate configurations that are 10% more resilient than configurations that only use static routes.

Second, we consider a setting in which the operator wants to generate configurations which reduce the number of policy violations occurring under common link failure scenarios. We call this notion *policy-resilience*. Since synthesizing policy-resilient configuration is very difficult in the general case, we focus on a restricted class of policies: for traffic class (src-dst subnet pair), we allow the operator to specify a set of waypoints that packets must traverse before reaching their destination. We modify our linear constraints to ensure that at least two paths that traverse the waypoints have path cost (e.g., OSPF path cost) lower than any path not going through a waypoint, thus providing 1-resilience under failure. Using this, Zeppelin can generate configurations that are 140% more policy-resilient than configurations generated with our first technique (i.e., reduces policy violations under failures by 140%).

Finally, we consider a setting where the operator can specify bounds on the number and sizes of domains her control plane is organized into, and the ability to assign routers to different domains. We present a stochastic search technique that leverages this flexibility to look for ways to assign routers to different domains so that the synthesized configurations have even higher resilience. Through this, Zeppelin can further improve the resilience of the configurations by 10%. We show that Zeppelin can be naturally used to bound or optimize different configuration metrics such as static routes and BGP configurations' size to improve network manageability.

Our experiments show that, thanks to its two-phase approach, Zeppelin synthesizes connectivity-resilient configurations for medium-sized datacenter topologies in < 10 *minutes* and policy-resilient configurations in under an hour. Notably, Zeppelin's performance is *2-3 orders of magnitude* faster than the state-of-art network configuration synthesis system SyNET [29], which uses a direct synthesis approach based on SMT.

Contributions.

- Zeppelin, a framework for that enforces policies in "traditional" (OSPF and BGP) networks by synthesizing highly-resilient router configurations. Zeppelin uses concrete paths to guide the synthesis instead of directly generating policy-compliant configurations.
- An algorithm for synthesizing policy-compliant configurations with few statically assigned routes. This algorithm yields configurations with high network connectivity even in the presence of link failures (§ 4.3).
- An algorithm for synthesizing policy-compliant configurations that is specialized for waypoint policies. This algorithm yields configurations that have high policy compliance even in the presence of link failures (§ 4.4.2).
- A stochastic search mechanism for finding partitions of the network into multiple routing domains which yield configurations with higher resilience (§ 4.5).
- An implementation of Zeppelin together with an evaluation of its algorithms for different workloads, and comparison with a state-of-the-art configuration synthesis tool (§ 4.6).

4.1 Overview

Before formally defining the problem tackled by Zeppelin and the algorithms, we present an overview of Zeppelin using an end-to-end example. Consider the hierarchical network in Figure 4.2 which is split into three OSPF domains, and these domains communicate using BGP. Host A talks to hosts B and C with the following policies: traffic to B must go through a firewall at router R_3, traffic to C goes through an IDS at R_5 and traffic to B and C do not share any links (We discuss the policies supported in detail in Section 4.2.2). Zeppelin synthesizes OSPF and BGP router configurations satisfying the policies. Illustrating our two-phase approach (Figure 4.1), Zeppelin first uses Genesis [86], a policy-enforcement framework for SDNs to find policy-compliant paths (indicated by the blue and red paths). Note that there exists multiple solutions for the given policies, we show how the control plane is synthesized using one of the solutions:

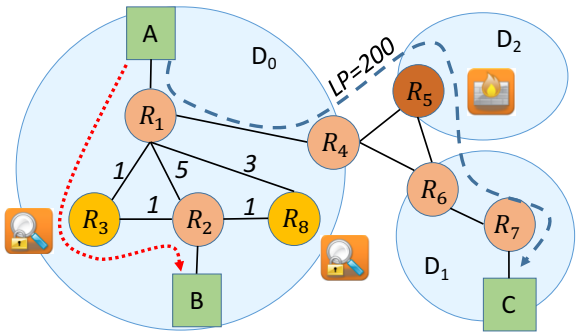


Fig. 4.2 Example demonstrating the two-phase synthesis approach. Genesis produces policy-compliant paths (red and blue), which are used as input by Zeppelin to synthesize the OSPF and BGP configurations.

- For traffic $A \rightarrow B$, the path is entirely inside a single OSPF domain, so Zeppelin has to find OSPF weights such that the shortest path is $R_1 \rightarrow R_3 \rightarrow R_2$. From the OSPF weights shown in Figure 4.2, we can see that the weight of path $R_1 \rightarrow R_3 \rightarrow R_2$ (2) is smaller than that of path $R_1 \rightarrow R_2$ (5).
- For traffic $A \rightarrow C$, the path traverses through multiple domains, so Zeppelin configures BGP such that routing across domains follows the Genesis path. In this case, we need traffic to go through a longer path in terms of AS path length; Zeppelin sets R_4's BGP local preference for route C from R_5 to a higher value (200), so that R_5 is preferred over R_6.
- Zeppelin can find better control planes in terms of how the network is split into domains. In this case, Zeppelin will suggest moving R_5 to domain D_1. This simplifies R_4's BGP configuration as the local preference entry for C is not required.
- With a backup firewall at R_8, Zeppelin automatically synthesizes resilient OSPF configurations for D_0 such that if either $R_1 - R_3$ or $R_3 - R_2$ link fails, traffic to B goes through the firewall at R_8.

Zeppelin is fundamentally different from state-of-art approaches for policy enforcement in legacy networks: Propane [11] generates BGP configurations from peering and path requirements, while Fibbing [89] uses fake advertisements to centrally control OSPF routing. SyNET [29] targets BGP and OSPF configuration synthesis, however, it does not cleanly tackle domain assignments or provide support for resilience. Zeppelin has more general policy coverage, supports automatic domain assignment (which is manual and difficult today), and can provide fault-tolerant configurations, all in one system.

Term	Description	Term	Description
$T = (V, L)$	Topology T of routers V and links L	$D(r_1, r_2)$	Shortest OSPF distance from r_1 to r_2
$\Theta(r) = \theta$	Router r belongs to domain θ	$G(\theta, \lambda) = \{g_1, g_2\}$	Set of exit gateway routers for λ in domain θ
$SR(r, \lambda) = \{r_1\}$	Static route $r \rightarrow r_1$ for destination λ	$LP(r, r', \lambda)$	Local preference at r to choose r' route to λ
$W(r_1, r_2)$	OSPF weight of link $r_1 \rightarrow r_2$	$IF(r_2, \lambda) = \{r_1\}$	r_2 will not advertise λ to r_1 using iBGP

Fig. 4.3 Different configuration parameters in Zeppelin

4.2 Problem Definition

In this subsection, we formally define the problems addressed in this chapter. We first describe the representation of different routing protocols; we model static routes, OSPF shortest-path routing, and BGP preference-based routing. Then, we define what it means for a configuration to meet a given set of policies and present the configuration synthesis problems we tackle. Finally, we formalize two resilience metrics called connectivity-resilience and policy-resilience; the goal of Zeppelin is optimizing these metrics.

4.2.1 Routing Model

We represent the physical router topology as a directed graph $T = (V, L)$, where V is the set of routers and $L \subseteq V \times V$ is the set of links. Throughout the chapter we assume T is fixed. We use the neighbour function $N(s) = \{s' \mid (s, s') \in L\}$ to denote the set of neighbour routers of s. A path $\pi = (r_1, r_2)(r_2, r_3) \cdots (r_{n-1}, r_n) \in L^*$ is a loop-free valid path if a router is not visited more than once, denoted by $valid(\pi)$. We also represent the path π as $r_1 \rightarrow r_2 \rightarrow \cdots \rightarrow r_n$. We write $l \in \pi$ when the path π contains the link l. We define $\Lambda \subset IP$ to denote the set of destination IP addresses; distributed protocols make forwarding decisions based on the destination address/subnet.

Hierarchical control planes partition the network into multiple connected components called domains. We define a router domain assignment function $\Theta : V \mapsto \mathbb{N}$ which maps each router to a domain (denoted by a number). We assume each domain uses OSPF as the intra-domain routing protocol and BGP as the inter-domain routing protocol. We summarize the notations in Figure 4.3.

Static Routes. These are statically configured next-hop routes with higher priority over those routes computed by OSPF and BGP.

Algorithm 3 BGP Best Path Selection at router r for λ

```
1:  procedure BESTPATH
2:      /* Each route γ of form (λ, local_pref, len, nexthop) */
3:      Υ_ebgp = ⋃ Route from external BGP neighbour for λ
4:      Υ_ibgp = ⋃ Route from internal BGP router r1: r ∉ IF(r1, λ)
5:      Υ = Υ_ebgp ∪ Υ_ibgp

6:      /* Prefer the path with the largest local preference */
7:      Υ_lp = {γ ∈ Υ | ∀γ1 ∈ Υ. γ.local_pref ≥ γ1.local_pref}
8:      if |Υ_lp| = 1 then
9:          γ_best = γ ∈ Υ_lp
10:     else
11:         /* Prefer the path with the smallest AS Path length */
12:         Υ_as = {γ ∈ Υ_lp | ∀γ1 ∈ Υ_lp. γ.len ≤ γ1.len}
13:         if |Υ_as| = 1 then γ_best = γ ∈ Υ_as
14:     if γ_best ∈ Υ_ebgp then
15:         R_bgp(r, λ) = γ_best.nexthop
16:         Redistribute γ_best to OSPF domain
17:     else
18:         /* Traffic for λ will not exit through r */
19:         R_bgp(r, λ) = ⊥
```

OSPF. In Open Shortest Path First (OSPF) routing, traffic from a router r to a destination λ is forwarded across the shortest path from r to λ. We define the OSPF routing function $\mathfrak{R}_{ospf} : V \times \Lambda \mapsto 2^V$ so that $\mathfrak{R}_{ospf}(r,\lambda)$ describes the next-hop for router r which lies on the shortest path for a destination λ. Suppose λ is directly connected to a router r_λ in the same OSPF domain as r.

$$\mathfrak{R}_{ospf}(r,\lambda) = \{r_1 \mid r_1 \in N(r) \ \wedge \ \forall r_2 \in N(s).\ W(r,r_1) + D(r_1,r_\lambda) \leq W(r,r_2) + D(r_2,r_\lambda)\}$$

BGP. BGP is a path-vector inter-domain routing protocol that connects different domains (or ASes). BGP routers can be configured with a local preference (LP) to assign higher priorities to specific routes for a particular destination. The Internal BGP (iBGP) protocol is used to exchange external BGP routes among BGP routers belonging to the same domain. Informally, iBGP propagates local preferences of the routes to all the BGP routers within the same domain. We can configure iBGP filters (IF) to prevent a router from advertising a route.

Algorithm 3 defines the BGP routing function based on the best path selection at each BGP router. A BGP router receives eBGP and iBGP routes; it chooses a route with highest local preference, and if there is a tie, it chooses the route that traverses fewer domains [74]. We assume all ties are broken using these criteria. After route selection, if a eBGP route is preferred, the route is redistributed to OSPF. Therefore, for a domain θ, only BGP routers which redistribute routes for destination λ to OSPF belong to the gateway router set $G(\theta,\lambda)$.

We define a partial BGP routing function $\mathfrak{R}_{bgp} : V \times \lambda \mapsto V$ such that $\mathfrak{R}_{bgp}(r, \lambda)$ describes the next-hop BGP router for the BGP router r when forwarding traffic for a destination λ.

OSPF+BGP+SR. We now describe how routing happens in hierarchical networks with all the previously described protocols used together. We express the complete network configuration C as a tuple $(T, \Theta, W, LP, IF, SR)$. We define the routing function $\mathfrak{R}^C : V \times \Lambda \mapsto 2^V$ so that $\mathfrak{R}^C(r, \lambda)$ describes the next-hop router for the router r when forwarding traffic for a destination λ. The priority order based on administrative distance [25] is SR $>$ BGP $>$ OSPF. Piecing together the different routing protocols and their interactions, the routing function $\mathfrak{R}^C$ for the hierarchical domain network configuration C is defined as follows:

$$\mathfrak{R}^C(r,\lambda) = \begin{cases} SR(r,\lambda) & \text{if } SR(r,\lambda) \neq \emptyset, \\ \mathfrak{R}^C_{bgp}(r,\lambda) & \text{if } r \in G(\Theta(r),\lambda), \\ \mathfrak{R}^C_{ospf}(r,\lambda) & \text{otherwise.} \end{cases}$$

Induced paths. We assume a finite set of packet classes $PC = \{0, \ldots, C_{pc}\}$ and map each reachability policy that requires the existence of a path between two endpoints to a unique integer in PC. The rest of the policies specify properties on these paths. Each packet class pc is associated with a tuple $(s_{pc}, d_{pc}, \lambda_{pc})$ which specifies the path from s_{pc} to d_{pc} for destination IP λ_{pc}.

Definition 2 (Induced Paths)**.** *Given a configuration C, the set of paths induced by the configuration C for the packet class pc:*

$$\mathfrak{P}^C(pc) = \{\pi = (u_1, v_1) \ldots (u_n, v_n) \mid valid(\pi) \wedge u_1 = s_{pc} \wedge v_n = d_{pc} \wedge \forall i.\ v_i \in \mathfrak{R}^C(u_i, \lambda_{pc})\}$$

Given a set of packet classes PC, the set of paths induced by C is defined as $\mathfrak{P}^C(PC) = \cup_{pc \in PC} \mathfrak{P}^C(pc)$.

4.2.2 Policy Support

One of the foremost tasks in network management in enterprise and multi-tenant datacenters—i.e., where different entities ("tenants") share the datacenter's infrastructure (compute, network etc.)—is programming networks to forward traffic in a manner consistent with user- and application-induced high-level policies for performance and security considerations. Unlike SDN, "traditional" networks are heterogeneous and run different kinds of protocols, and thus, operators require support to enforce and/or optimize different configuration structure and properties. We classify Zeppelin's policy support into two categories: (1) *path policies*:

Policy	Description
Reachability	There is a path from router s to router t for subnet λ
Reachability + Waypoints	The path from s to t for destination λ traverses a set of waypoints
Isolation	Paths of two reachability policies do not share links
Traffic Eng.	Minimize total/max link utilization

(a) Path Policies

Policy	Description
Count	No. of OSPF domains: $c_1 \leq N_D \leq c_2$
Domain Size	Limit number of routers in a domain: $l \leq ds \leq u$
BGP Enable	Enable BGP on routers B (hardware constraint)
Static Route sc	Limit number of static routes: $sc < SC$
BGP Config. Overhead bc	Upper bound the number of local preference entries and iBGP filters $bc < BC$
Cost Minimization	Minimize user-defined cost $expr(sc, bc)$

(b) Configuration Policies

Fig. 4.4 Zeppelin Policy Support

high-level intents on paths of different traffic classes, and (2) *configuration policies*: low-level intents on the deployed router configurations.

Figure 4.4a describes the common path policies supported by Zeppelin. Operators can specify policies in a declarative manner using a high-level policy language. Given a set Ψ of path policies, we say that a set of paths Π is policy-compliant with respect to Ψ, denoted $\Pi \models \Psi$, if the paths in Π satisfy all the policies in Ψ [86].

Reachability and Waypoints. This policy enables network communication between pairs of a tenant's virtual instances (VM), applications, or hosts. The tenant may wish that the flow between two of her end hosts, or from another tenant, must traverse specific middleboxes, which we also refer to as "waypoints".

Isolation. Tenants may require various Quality-of-Service (QoS) or security guarantees since the underlying infrastructure is shared among tenants. In the extreme, a tenant could require that her flows are not affected in any manner by any other tenant by strictly isolating the path of the tenant's flows from others' flows.

Traffic Engineering. While support for the above policies can be used to satisfy tenant requirements, network operators need to carefully manage constrained resources. Operators may also want to balance load on their network infrastructure. This is often done by optimizing a network-wide objective such as total or maximum utilization of network links due to traffic induced by all tenants.

The high-level path policies are enforced by different low-level configuration constructs pertaining to routing protocols like OSPF and BGP, and ideally, the operator requires support to impose constraints on the structure and properties of the deployed configurations. Figure 4.4b describes all the configuration policies supported by Zeppelin for hierarchical control planes.

OSPF Domains and Sizes. We support a policy which restricts how many OSPF domains a configuration might have. This policy is useful for avoiding situations resulting in too many domains in the hierarchical split which can be difficult to administer. Another policy allows the operator to bound the size of each OSPF domain because OSPF does not scale gracefully with network size.

Resource Constraints. Certain routers may not be suited to run BGP due to resource constraints. Thus, the operator can specify what set of routers $B \subseteq V$ are BGP-compatible.

Configuration Metrics. Zeppelin provides ways to specify upper bounds on the number of static routes *sc* and BGP configuration overhead *bc* which we define as the sum of local preference entries and iBGP filters. While static routes are a powerful tool, it is desirable to limit their usage since they can lead to undesired routing behaviors such as routing loops. Similarly, it is desirable to limit the number of BGP configurations since they increase the complexity of the network. Zeppelin can also try to minimize certain expressions over the quantities *sc*, and *bc*—e.g., $max(sc, bc)$.

4.2.3 Synthesis of policy-compliant configurations

We now define when a configuration C is policy-compliant, present our synthesis problems, and introduce two definitions of resilience we will use throughout the chapter.

Definition 3 (Policy-Compliance and Synthesis)**.** *Given a set of path policies* Ψ *and a set of configuration policies P, a configuration C is policy-compliant with* (Ψ, P)*, written as* $C \models (\Psi, P)$*, if the set of induced paths satisfies* Ψ*—i.e.,* $\mathfrak{P}^C(PC) \models \Psi$*—and C satisfies the policies in P. The* configuration synthesis problem *is to find, given* Ψ *and P, a configuration C that is policy compliant with* (Ψ, P)*.*

Our approach will proceed in two phases, one of which solves the following sub-problem.

Definition 4 (Path-Compliance and Synthesis)**.** *Given configuration policies P and a set of paths* Π *over packet classes PC, a configuration C is path-compliant with* (Π, P)*, if* $\mathfrak{P}^C(PC) = \Pi$ *and C satisfies the policies in P. The* path-compliance synthesis problem *is to find, given P and* Π*, a configuration C that is path-compliant with* (Π, P)*.*

1-Resilience metrics. One of the main goals of this work is to generate configurations that are highly resilient to failures. In this work, we focus on resilient configurations for single link failures. [1] For a packet class *pc* and configuration *C*, we define the set of links *affecting*

[1] We do not consider multiple link failures as they occur with lower probability [43].

pc as $\mathbb{L}(pc) = \{l \mid \exists \pi \in \mathfrak{P}^C(pc).\ l \in \pi\}$. For a single failure of a link l, a packet class pc is affected if only if $l \in \mathbb{L}(pc)$. Given a configuration $C = ((V,L),\Theta,W,LP,IF,SR)$ and a link $l \in L$, we write C_l to denote the configuration $C = ((V,L \setminus \{l\}),\Theta,W,LP,IF,SR)$ in which the link l has been removed from the topology. We present two different resilience metrics we will use in the following subsections.

The first metric describes how good a given configuration is at preserving policy-compliance under an arbitrary single link failure. The score measures, for each policy pc, what percentage of the links affecting pc causes a policy to still hold when failing.

Definition 5. *The* policy-resilience *score of a configuration* $C = ((V,L),\Theta,W,LP,IF,SR)$ *is defined as:*

$$PR(C) = \frac{\sum_{\forall pc} |\{l \mid l \in \mathbb{L}(pc) \ \wedge\ C_l \models (\Psi_{pc}, P)\}|}{\sum_{\forall pc} |\mathbb{L}(pc)|}$$

where $C \models (\Psi_{pc}, P)$ *means that C is compliant for the set of policies* Ψ_{pc} *that affect packet class pc.*

In general, generating configurations with high policy-resilience is hard, since there may be only a few paths that satisfy a certain policy. Therefore, we also consider a second more relaxed metric, which describes how good a given configuration is at preserving connectivity (but not policy compliance) upon a single link failure.

Definition 6. *The* connectivity-resilience *score of a configuration* $C = ((V,L),\Theta,W,LP,IF,SR)$ *is defined as follows:*

$$CR(C) = \frac{\sum_{\forall pc} |\{l \mid l \in \mathbb{L}(pc) \ \wedge\ \mathfrak{P}^{C_l}(pc) \neq \emptyset\}|}{\sum_{\forall pc} |\mathbb{L}(pc)|}$$

Ideally, operators desire a resilience score of 1 (perfectly resilient for policy-compliance or preserving connectivity). However, achieving this objective is in practice difficult due to the increased number of configurations one needs to consider, and sometimes impossible due to the structure of the network and the nature of policies. In practice, Zeppelin can synthesize configurations with high resilience scores and operators can run it to generate different configurations until a threshold score is crossed.

4.3 From Policies to Connectivity-Resilient Configurations

We now present an algorithm for synthesizing distributed configurations that adhere to path policies like the one described in Figure 4.4a. In general, no existing tool for synthesizing

distributed configurations can handle this heterogeneous set of these policies—e.g., isolation—and, even less so, generate resilient configurations. Due to the complexity of the problem, we focus on synthesizing configurations that are *policy-compliant* and have high *connectivity-resilience*.

4.3.1 A two-phase approach

Existing approaches to configuration synthesis try to directly generate configurations from the given policies [29], but a direct approach leads to scalability issues as well as limitation in the set of supported policies. Instead of directly generating configurations from policies, Zeppelin uses a two-phase approach: first it generates policy-compliant paths and then it synthesizes distributed configurations that realize these paths and have high connectivity resilience.

For the first phase, we use Genesis [86], a network management system which synthesizes SDN forwarding tables enforcing some input policies. Some of the policies supported by Genesis are specified in Figure 4.4a. Given a set of policies, Genesis generates a set of constraints so that solutions to these constraints are forwarding tables that can be used to extract the policy-compliant paths.

However, we need to modify the Genesis constraints to integrate with Zeppelin. Legacy protocols only support destination-based forwarding, unlike OpenFlow switches [63]. A router cannot forward a subnet's traffic to different routers based on source (or other packet headers). Thus, we add additional constraints to ensure the paths generated by Genesis follow destination-based forwarding: paths obtained from Genesis for a destination will form a directed tree.

Let us now look at how we solve the path-compliance problem. Formally, we are given a set of paths Π, a topology $T = (V, L)$, a domain-assignment function Θ, and we want to find functions of the configuration $C = (\Theta, W, LP, IF, SR)$ such that $\mathfrak{P}^C(PC) = \Pi$. In a software-defined OpenFlow [63] network, we can program switch rules to forward traffic to a particular next-hop switch. Therefore, a trivial solution to the path-compliance problem in SDNs is to install forwarding rules along the policy-compliant paths [86]. Static routes provide the functionality of specifying the next-hop router, thus, a solution to the path-compliance problem is to simply add a static route to $SR(\lambda)$ for every link l in set of the paths Π. Similar to static routes, Fibbing [89] used fake advertisements to achieve fine-grained control over OSPF forwarding.

Under no failures, the trivial solution of using static routes ensures policy-compliance, as traffic follows the paths provided by Genesis. However, this solution uses more static routes than necessary and, in the presence of failures, static routes may create routing loops that

make certain connections unreachable, hence reducing connectivity-resilience. Thus, we set our goal to synthesizing configurations which are policy-compliant when there are no network failures, and improving connectivity-resilience by using fewer static routes.

We first show how to synthesize path-compliant intra-domain configurations and then extend to inter-domain configurations.

4.3.2 Synthesizing Path-compliant Intra-domain Configurations

In this subsection, we show how to synthesize path-compliant intra-domain configurations that use few static routes. The problem of optimally placing static routes is NP-hard (Theorem C.1.1), therefore, we propose a non-optimal greedy strategy. We first show how to solve the problem when there exists a solution with no static routes and then extend our technique to greedily add static routes when needed. We use efficient off-the-shelf linear programming solvers for solving the system of constraints which are presented in the following subsections.

4.3.2.1 Intra-domain Synthesis without Static Routes

In the absence of static routes, OSPF routers use edge weights to choose the shortest weighted path for each pair of endpoints. The problem of synthesizing the weight function W that realizes an input set of paths Π is a variation of the so-called *inverse shortest path* problem [19]. For a destination IP λ, we call ξ_λ the directed tree of T obtained by only keeping the nodes and edges that are traversed by paths in Π for λ; the root of the tree is the destination router connected to λ. This destination tree property is due to the modifications to Genesis to support OSPF's destination-based forwarding. We define $\Delta = \{\xi_\lambda \mid \lambda \in \Lambda\}$ to be the set of all destination trees.

Given a set of input paths Π, Zeppelin generates a set of linear constraints to find weights for each directed link in the domain. The constraints use the variable $W(r_1, r_2)$ and $D(r_1, r_2)$ to denote the weight and shortest distance respectively. We add the equation $D(s,s) = 0$. Equation (4.1) guarantees that $D(s,t)$ is smaller or equal to the shortest distance from s to t.

$$\forall s,t.\ \forall r \in N(s).\ D(s,t) \leq W(s,r) + D(r,t) \tag{4.1}$$

Intuitively, the shortest path connecting s to t must traverse through one of the neighbor routers of s, and thus, distance can be defined inductively as the shortest among distances from the neighbors.

For each destination tree $\xi_\lambda \in \Delta$, we add equations to ensure that the input paths with destination λ are the shortest ones. Notice that, if a path π is the shortest path between its endpoints, every subpath of π also has to be the shortest path between its endpoints. For

each tree ξ_λ, we define the neighbor $N_{\xi_\lambda}(s)$ to denote the next-hop neighbor of router s in the destination tree ξ_λ. We denote the router directly connected to λ (the root of the tree ξ_λ) by R_λ.

For a path π, $\sum_\pi W$ denotes the sum of weights of all links in π. The following equation ensures that, for any node s in ξ_λ, the path $(s,r_1)\cdots(r_n,R_\lambda)$ in ξ_λ is the *unique shortest path*:

$$\forall s \in \xi_\lambda \,.\, \forall r \in N(s) \setminus N_{\xi_\lambda}(s) \,.\, \sum_{(s,r_1)\cdots(r_n,R_\lambda)} W < W(s,r) + D(r,R_\lambda) \tag{4.2}$$

These constraints guarantee that the sum of the weights belonging to the path from s to R_λ in ξ_λ is strictly smaller than any other path that goes to R_λ via a node n'. Note that, while $D(r,R_\lambda)$ can be smaller than the actual shortest distance from r to R_λ, $D(r,R_\lambda)$ is used to upper bound the sum of edge weights in ξ_λ, and thus, the solution to the edge weights will ensure paths in ξ_λ are the shortest.

Soundness and Completeness. OSPF synthesis is sound—i.e., weights satisfying constraints (4.1) and (4.2) will ensure all paths in Π are the unique shortest paths. OSPF synthesis is complete as well—i.e., if all paths in Π are the unique OSPF shortest paths, the weights will satisfy the constraints.

4.3.2.2 Intra-domain Synthesis with Static Routes

If the system of equations presented in §4.3.2.1 admits a solution, the values of the $W(s,t)$ variables are the weights we are trying to synthesize, otherwise the intra-domain synthesis problem cannot be solved without static routes. Consider the paths in Figure 4.5 between s and t for λ_1 and λ_2. There exists no solution as the following is required for λ_1: $W(s \to r_1 \to t) < W(s \to t)$, and vice-versa for λ_2. To steer the traffic for a particular destination to a next-hop router not on the shortest path, we install a static route.

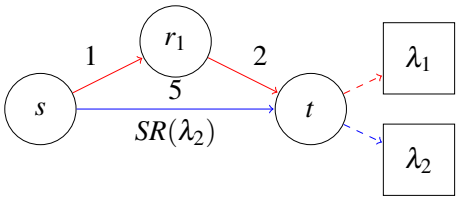


Fig. 4.5 Example illustrating the use of static routes.

Since the problem of minimizing the number of static routes is NP-hard, we opt to place static routes using a best-effort approach. Our algorithm starts by trying to synthesize a solution that does not use static routes using the equations proposed in §4.3.2.1. In the case of a failure, the algorithm uses the "proof of unsatisfiability"—i.e., the unsatisfiable core—generated by the constraint solver to greedily add a small set of static routes. Certain equations are eliminated to model the added static routes and the approach is repeated until a solution is found.

Algorithm 4 OSPF-SR Synthesis

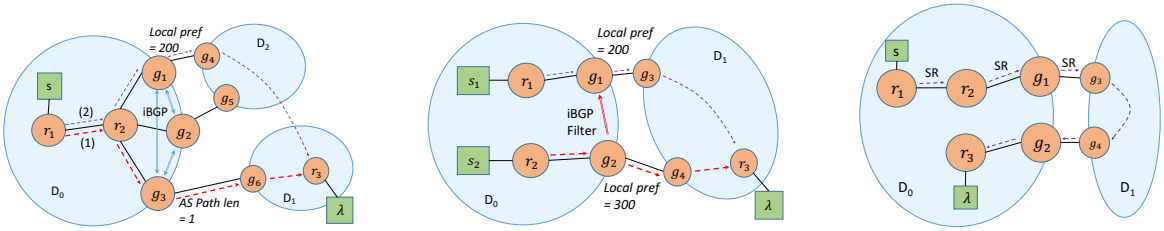
```
1: procedure OSPF_SYNTH(Π)
2:     Ψ_D : Distance constraints (4.1)
3:     Ψ_S : Shortest path constraints generated for Π (4.2)
4:     Ψ = Ψ_D ∪ Ψ_S
5:     while Ψ is unsat do
6:         Extract unsat core uc from LP Solver
7:         Pick random static route sr(s,t,λ) from uc
8:         Add sr to static routes SR
9:         Ψ = Ψ \ Ψ_S(s,λ)
10:    Obtain W from solution model of Ψ
11:    return Configuration C(W,SR)
```

Intuitively, an unsat core or IIS (Irreducible Inconsistent Subsystem) [23] is a subset of the input constraints such that, if all constraints except those in the IIS are removed, the resulting set of linear equations is still unsatisfiable. Moreover, the set is irreducible—i.e., removing any one constraint from the IIS produces a satisfiable set of constraints. In our case, an IIS cannot consist of only constraints from Equation (4.1) as these constraints admit a trivial solution with all variables set to 0. Therefore, an IIS must contain some constraint of the form given in Equations (4.2). Let us denote this set of constraints as $\Psi_S(s,\lambda)$:

$$[\sum_{l_i=(r_1,r_2)} W(r_1,r_2) < W(s,r) + D(r,R_\lambda)] \in \Psi_S(s,\lambda)$$

This constraint is added to ensure that router s forwards to traffic to next-hop $N_{\xi_\lambda}(s)$ and not some neighbor router r, based on OSPF weights, but the path through r is causing the unsatisfiability. To remove this inequality from the set of constraints, we add the static route $(s, N_{\xi_\lambda}(s))$ to $SR(\lambda)$. As a result of adding a static route, Zeppelin removes the $\Psi_S(s,\lambda)$ constraints as router s will forward λ traffic to next-hop $N_{\xi_\lambda}(s)$ irrespective of the OSPF distances to the destination; the unsatisfiability caused by this IIS is eliminated. However, the new set of constraints may still be unsatisfiable due to other IISes. We repeat the procedure and add static routes until we obtain a satisfiable set of constraints. In each iteration, there can be more than one way to place a static route and Zeppelin picks one randomly.

Soundness and Completeness. OSPF synthesis with static routes is sound—i.e, the configuration $C(W,SR)$ will induce the paths Π provided by Genesis. As far as completeness goes, Zeppelin is guaranteed to find a solution if there are no constraints on the number of static routes (use static routes along each path). If there is a configuration policy upper bounding the number of static routes, the synthesis procedure is not guaranteed to find a compliant configuration, if one exists. Since there can only be finitely many solutions for paths and possible SR assignments, we are eventually guaranteed to find a solution by repeating Genesis and OSPF-SR synthesis multiple times and choosing different solutions.



(a) Single BGP Gateway (b) Multiple BGP Gateways (c) Static Routes

Fig. 4.6 Examples of BGP and static route configurations for inter-domain routing given domain assignment Θ.

4.3.3 Synthesizing Path-compliant Inter-domain Configurations

Next, we extend our algorithm to solve the path-compliance problem in the presence of multiple domains. Our algorithm determines the BGP configuration and static routes to ensure that the paths will traverse the correct domains, and new constraints required to OSPF synthesis for the inter-domain scenario.

Configuring BGP and Static Routes. In this subsection, we describe how to configure BGP routers so that the routes chosen by BGP (and redistributed into OSPF) can yield the policy-compliant paths Π given as input. Note that, the BGP $\rightarrow$ OSPF redistribution can sometimes lead to explosion of OSPF routing tables. However, in our datacenter setting, BGP is used in a restricted sense for management ease and BGP routers are not connected to the external internet directly, thus, only internal prefixes are redistributed.

To enforce the policy-complaint paths, we need to take into account three different cases, which are illustrated in Figure 4.6. In the first case, path to destination λ exits the domain through a single BGP gateway, illustrated in Figure 4.6(a). For path (1)—i.e., the red one—to be the chosen path, it is already on the shortest AS path, therefore, the gateway for λ in domain D_0 is g_3, i.e., $G(0,\lambda) = g_3$. However, if path (2)—i.e., the purple one— is the path from Genesis, it has to traverse more domains. Thus, we set local preference for route for λ to g_1 to 200, i.e., $LP(g_1, g_4, \lambda) = 200$. After iBGP route distributions, the gateway route for λ will be through g_1.

In the second case, two paths from different sources to destination λ exit the domain through two different gateways. In Figure 4.6(b), the path from r_1 to λ exits domain D_0 through gateway router g_1, while the path from r_2 to λ exits through g_2. In this case, both these routes need to be redistributed to the OSPF domain. By assigning local preferences such that $LP(g_1, g_3, \lambda) < LP(g_2, g_4, \lambda)$, g_2 will redistribute a route for λ to the OSPF domain and the traffic from r_2 will correctly exit through g_2. To prevent the traffic from r_1 to exit through g_2, we add an iBGP filter for the connection $g_2 \rightarrow g_1$ for destination IP λ. Thus, g_1 will redistribute the route it received from g_3. For this example, $G(0,\lambda) = \{g_1, g_2\}$.

Finally, we describe a case in which standard BGP cannot enforce the desired path, therefore requiring the use of static routes. Consider the path for destination λ shown in Figure 4.6(c) which has a domain loop in the path ($D_0 \rightarrow D_1 \rightarrow D_0$), thus, this path cannot be enforced by conventional BGP routing. To circumvent this issue, Zeppelin uses static routes (illustrated in Figure 4.6(c)). To find the minimal number of static routes to be used in such cases, we find the longest path suffix that has no domain loops (from the start of this path fragment, BGP+OSPF can be used for routing), and add static routes from the source to the start of this path suffix.

Modified OSPF Synthesis. When BGP routes for destination λ are redistributed by multiple gateways to an OSPF domain, Zeppelin uses a modified OSPF synthesis algorithm from §4.3.2. In case of multiple gateways, an OSPF router will choose the closest BGP gateway in terms of OSPF distance for λ. For instance, a router r will choose gateway g_1 over g_2 if the OSPF distance from r to g_1 is strictly lesser than the distance from r to g_2. We add new constraints on OSPF weights (details omitted for brevity) to ensure that any router on the path is closest to the gateway in the path.

4.4 From Waypoint Policies to Policy-Resilient Configurations

The algorithm presented in the previous subsection generates configurations which guarantees enforcement of complex policies under no failure scenarios, and at the same time, achieves good connectivity-resilience by using fewer static routes. However, the generated configurations do not provide any guarantees as to whether the policies are satisfied when failures occur in the network. In this subsection, we present a different algorithm that, for a restricted set of policies, can synthesize configurations with high *policy-resilience*. In particular, we focus on *waypoint policies* which play a crucial role in enterprise and datacenter networks for security, performance, and auditing.

4.4.1 Problem Setup

We consider policies of the form λ : `s` » $\mathbb{W}$ » `t` where λ is a destination IP subnet, s and t are the source and destination routers respectively, and $\mathbb{W} \subseteq V$ is a set of routers called the *waypoint set*. Each policy is mapped to a packet class and we say that a configuration complies to a policy for pc if all induced paths in $\mathfrak{P}^C(pc)$ traverses some waypoint $w \in \mathbb{W}$. Here, $\mathbb{W}$ can be the set of physical replicas of, for example, a firewall. Given a set of such waypoint policies, our goal is to find a policy-compliant configuration $C = (T, \Theta, W, LP,$

IF, SR) with high policy-resilience—i.e., the configuration must be waypoint-compliant under single-link failures.

We present an algorithm that solves this problem when the network has a single domain; when the network has multiple domains, we assume that all waypoints belong to the same domain and use our new algorithm in the domain which contains the waypoints. This assumption can be realized through Network Function Virtualization [39, 68], which provides waypoint replication and flexible waypoint placement in the network.

4.4.2 Intra-domain Policy-Resilient Waypoint Compliance

Genesis provides support for link-isolation, which can be used in conjunction with the waypoint policy to generate multiple disjoint paths for a particular destination which are waypoint-compliant. For our case of 1-resilience, Zeppelin uses Genesis to generate two link-disjoint paths, where each path traverses through one of the waypoints in $\mathbb{W}$. Link-disjointedness ensures that a single link failure will not disable both these paths and we could guarantee policy-resilience by ensuring that, under any link failure, traffic to λ traverses one of these two paths. However, enforcing the control plane to always route through one of the two paths is difficult and overly restrictive. Instead, we relax our constraints to guarantee that the two paths are shorter than any non-compliant path in the network. The existence of two such paths still guarantees that traffic under any single link failure, traverses a waypoint.

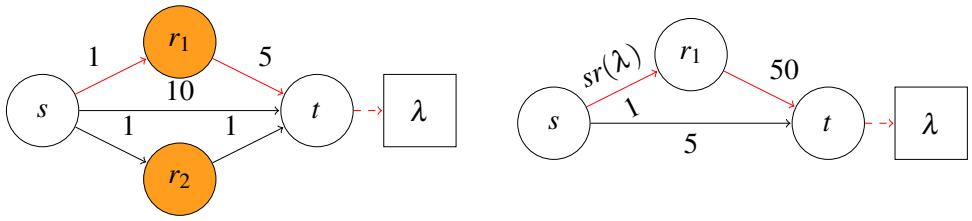
4.4.2.1 Waypoint-Compliance Constraints

We now show how to generate constraints that guarantee that a single waypoint-compliant path is shorter than any path which does *not* traverse a waypoint. We define $D(s,t,\mathbb{W})$ to be the distance between s and t for paths that *do not* traverse any waypoint $w \in \mathbb{W}$. We call $D(s,t,\mathbb{W})$ the *non-waypoint distance*. We add constraints to represent these distances by considering a network topology where all the waypoints $w \in \mathbb{W}$ are removed:

$$\forall s,t \in V \setminus \mathbb{W}.\ \forall r \in N(s) \setminus \mathbb{W}.\ D(s,t,\mathbb{W}) \leq W(s,r) + D(r,t,\mathbb{W}) \tag{4.3}$$

Using these equations, $D(s,t,\mathbb{W})$ is upper bounded by the actual shortest non-waypoint distance from s to t.

Given a destination tree ξ_λ, we can use non-waypoint distances to enforce waypoint-compliance. Consider a router in r such the waypoint $w \in \mathbb{W}$ is downstream to r in ξ_λ—i.e., there exists a path from r to w in ξ_λ. The path from r must traverse through a waypoint in $\mathbb{W}$. If the sum of weights of the edges of the path in the tree $r \rightarrow^{+}_{\xi_\lambda} R_\lambda$ is strictly smaller than the



(a) Waypoint-compliant edge weights (b) Routing Loop

Fig. 4.7 Example of waypoint-compliant edge weights and example of a routing loop caused by a static route.

non-waypoint distance from r to R_λ, then the path from r to R_λ is guaranteed to traverse a waypoint $w \in \mathbb{W}$. These constraints are expressed as:

$$\forall r' \in N(s) \setminus N_{\xi_\lambda}(r). \sum_{r \to^+_{\xi_\lambda} R_\lambda} W < W(r,r') + D(r', R_\lambda, \mathbb{W}) \tag{4.4}$$

These equations do *not* guarantee that router r will forward traffic to $N_{\xi_\lambda}(r)$. Consider the example in Figure 4.7(a) where traffic for λ must traverse through one of the waypoints in $\{r_1, r_2\}$. In this case, Genesis provided the path $s \to r_1 \to t$, but the shortest path is $s \to r_2 \to t$ which is waypoint-compliant.

Soundness and Completeness. With no static routes, if edge weights satisfy constraints (4.3) and (4.4), then paths for packet classes will traverse one of the waypoints. However, these constraints do not ensure completeness—i.e., a solution could be waypoint-compliant while violating some of the constraints, specifically constraint (4.4): while the weight of the Genesis path may be greater than the non-waypoint distance, the actual shortest path could be waypoint-compliant.

4.4.2.2 Avoiding Routing Loops

Algorithm 5 presents the OSPF synthesis algorithm with static routes. If one of the constraints in Equation (4.4) is part of an unsatisfiable core, Zeppelin adds the static route $(r, N_{\xi_\lambda}(r))$ to $SR(\lambda)$ to eliminate the equations at router r. Constraints (4.4) do not guarantee that the path generated by Genesis is indeed the shortest path, so adding static routes can lead to undesired behaviors like routing loops. Consider the example configuration shown in Figure 4.7(b). If link $r_1 \to t$ fails, traffic will oscillate between s and r_1 due to SR and OSPF and finally be dropped.

Whenever Zeppelin adds a static route to resolve an unsat-core, it removes the constraints pertaining to the static route and *adds* new constraints to prevent routing loops. Suppose

Algorithm 5 OSPF Waypoint Synthesis with Static Routes

procedure OSPF_W_SYNTH(Π)
  Ψ_D : Distance (4.1) and non-waypoint distance constraints (4.3)
  Ψ_W : Waypoint constraints for Π (4.4)
  $\Psi_R = \emptyset$: Routing Loop avoidance constraints (4.5)
  $\Psi = \Psi_D \cup \Psi_W \cup \Psi_R$
  while Ψ is *unsat* **do**
    Extract unsat core *uc* from LP Solver
    Pick static route $sr(sr_1, sr_2, \lambda)$ from *uc* constraints
    Add *sr* to static routes *SR*
    $\Psi = \Psi \setminus (\Psi_W(sr_1, \lambda) \cup \Psi_R(sr_1, \lambda))$
    $\Psi = \Psi \cup \Psi_R(sr_2, \lambda)$
  Obtain W from solution model of Ψ
  return Configuration $C(W, SR)$

Zeppelin added a static route (sr_1, sr_2) for destination λ where $sr_2 = N_{\xi_\lambda}(sr_1)$ (we do not add static routes on any other links except ξ_λ). Zeppelin needs to ensure that, for any router $r \in \xi_\lambda$ that does not lie in the downstream path $sr_2 \rightarrow^+_{\xi_\lambda} R_\lambda$, the shortest path from sr_2 to R_λ does not traverse through r. For e.g., in Figure 4.7(b), there is a routing loop because the shortest path from r_1 to t goes through upstream router s.

Formally, Zeppelin adds constraints to ensure that the weight of path $sr_2 \rightarrow^+ R_\lambda$ is strictly smaller than any path from sr_2 that traverses a router r' not in the downstream path from sr_2:

$$\forall r' \in \xi_\lambda .\ sr_2 \not\rightarrow^+_{\xi_\lambda} r'.\ \sum_{sr_2 \rightarrow^+ R_\lambda} W < D(sr_2, r') + D(r', R_\lambda) \tag{4.5}$$

If one of these constraints is part of an unsatisfiable core, then there is a routing loop caused from sr_2. Zeppelin rectifies this by adding a static route $(sr_2, N_{\xi_\lambda}(sr_2))$ to $SR(\lambda)$. Algorithm 5 describes the unsat-core learning algorithm for waypoint-compliance.

Soundness and Completeness. Algorithm 5 is sound—i.e., for each packet class, there is a path which goes to one of the waypoint in the set. Algorithm 5 is not complete—i.e., not guaranteed to find C with a given bound on number of static routes.

4.4.2.3 Configurations for Two Paths

To guarantee policy-resilience, Zeppelin uses Genesis to generate two waypoint-compliant edge-disjoint paths π_1 and π_2 from s to t and adds the following constraints to ensure that the

weights of both paths are strictly shorter than the non-waypoint distance for λ.

$$\sum_{\pi_1} W < D(s,t,\mathbb{W}) \ \wedge \ \sum_{\pi_2} W < D(s,t,\mathbb{W}) \tag{4.6}$$

Notice that the constraints do not enforce an order between the weights of π_1 and π_2. Since the paths are edge-disjoint, if a single link fails, at most one of the paths is affected. For a OSPF configuration with no static routes, all paths for a policy will be waypoint-compliant under failures.

Soundness and Completeness. If there are no static routes, OSPF weights satisfying constraints (4.3), (4.4) and (4.6) ensure waypoint-compliance under any arbitrary single link failure. For completeness, the same observations as in subsection 4.4.2.1 also apply to this case.

4.4.2.4 Resilience with Static Routes

Providing policy-resilience with static routes is challenging because static routes do not react to failures. When static routes are used, we relax our definition of waypoint-compliance to ensure at least one of the induced paths for the policy traverses the waypoint. [2] We now present our approach to providing policy-resilience with static routes.

Zeppelin only adds static routes on the paths obtained from Genesis. Consider the example in Figure 4.8 where $\pi_1 = s \to r_0 \to r_1 \to t$ and $\pi_2 = s \to r_2 \to t$ are two edge disjoint paths connecting the routers s to t. In this example, π_1 is the path selected by the configuration and it uses a static route for $r_0 \to r_1$. Ideally, we want to ensure that if any link on π_1 fails, then the network must switch over to π_2. When link $r_1 \to t$ fails, router s forwards to r_0 (path weight 5: $s \to r_0 \to r_2 \to t$) over r_2 (path weight 6: $s \to r_2 \to t$). Because of the static route at r_0, traffic is forwarded to r_1 which sends it back to r_0 through r_3, causing a routing loop.

The problem is caused by the fact that π_1—i.e., the path chosen by the configuration—contains a static route. Upon a failure, even though there exists another waypoint-compliant path, π_2, the configuration will not switch to it. To avoid this problem, Zeppelin synthesizes configurations that split traffic among the two paths.

[2] Can be realized by two mechanisms: 1) A router which has multiple routes for λ will split the traffic among these routes, and 2) The waypoint can mark certain fields in the packet header, and the network operator can employ simple edge-based checks at the destination (for e.g., in the hypervisor) to detect if a packet traversed the waypoint or not. Thus, only packets which traversed a waypoint-compliant path would be accepted and forwarded to the tenant.

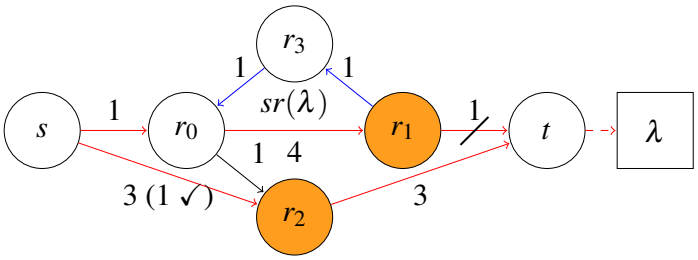


Fig. 4.8 Example non-resilient configuration for the red paths provided by Genesis for $\mathbb{W} = \{r_1, r_2\}$. If link $r_1 \to t$ fails, a routing loop is formed at $r_0 \to r_1 \to r_3 \to r_0$, no traffic reaches t. The configuration is 1-resilient waypoint-compliant when the $s \to r_2$ weight is set to 1.

However, simply adding the constraint $\sum_{\pi_1} W = \sum_{\pi_2} W$ does not ensure the traffic will be split (see Figure 4.8), due to the presence of static routes in the path. Intuitively, a static route is added to override OSPF and take a longer path, thus $\sum_{\pi_1} W$ is not the actual shortest path in the network. Thus, we need to estimate the weight of the shortest paths from the source corresponding to π_1 and π_2 and equate them to split the traffic.

In our running example, the distance between s to t corresponding to π_1 is equal to $W(s,r_0) + D(r_0,t) = 1 + (1+1+1)$ where r_0 is the position of the first static route. From s to r_0, since there are no static routes in the path, the OSPF distance can be estimated using the sum of edge weights, while the distance from r_0 to t is estimated using $D(r_0,t)$. Concretely, given $\pi_1 = (s,r_1)(r_1,r_2)\ldots(r_m,t)$, let r_α denote the first router in the path which has a static route—i.e., $\forall i < \alpha.(r_i, r_{i+1}) \notin SR(\lambda)$ for destination IP λ. We estimate the actual shortest OSPF distance between s to t corresponding to π_1 as the sum of weights till r_α plus the distance from r_α to t.

To enable splitting amongst π_1 and $\pi_2 = (s,s_1)(s_1,s_2)\ldots(s_n,t)$, we would like to ensure that the estimated OSPF distance of π_2 (with first static route at s_β) is smaller or equal to the estimated OSPF distance corresponding to π_1, and vice-versa. A first attempt at doing so is to use the following constraint.

$$\sum_{(s,s_1)\ldots(r_{\beta-1},r_\beta)} W + D(r_\beta,t) \;\leq\; \sum_{(s,r_1)\ldots(r_{\alpha-1},r_\alpha)} W + D(r_\alpha,t) \tag{4.7}$$

By virtue of distance constraints (4.1), $D(r_\beta,t)$ is upper-bounded by the actual shortest distance between r_β and t. However, the distance constraints do not impose a lower bound on $D(r_\beta,t)$ and a constraint of the form $D(r_\beta,t) \leq E$, where E is some expression, will also not impose a lower bound on the value of $D(r_\beta,t)$. Instead this may result in lower D and incorrect W values. Hence, we need to use $\sum_{\pi_2} W$ (which is an upper bound) for the OSPF

distance corresponding to π_2:

$$\sum_{\pi_2} W \leq \sum_{(s,r_1)\ldots(r_{\alpha-1},r_\alpha)} W + D(r_\alpha, t) \tag{4.8}$$

Since we cannot exactly express the weights of the actual routes seen at the source router, the above constraint does not guarantee that traffic will be split amongst π_1 and π_2. Similar to the routing loop avoidance constraints, we add the above constraints lazily when a static route is added to one of the paths for λ. However, if one of these constraints is part of an unsatisfiable core, we cannot add a static route to eliminate this unsatisfiability. Zeppelin lazily eliminates these constraints from the system of equations depending on whether these constraints are part of an unsat-core which cannot be eliminated by a static route. Thus, the presented approach does not provide provable resilience guarantees. However, our experiments show that Zeppelin generates highly resilient configurations (§4.6.2).

4.5 Increasing Resilience through Domain Assignment

Many networks today have routing hierarchies which use a variety of administrative policies or division of responsibilities. For example, a campus network may be hierarchically divided into multiple routing domains (e.g., OSPF Areas) corresponding to different departments. These hierarchies are generated manually by operators—e.g., to determine which routers to include in a particular OSPF routing domains, how many such domains to create, how big to make each domain etc. Unfortunately, generating the hierarchical division is painstaking and error-prone, and making changes to the hierarchies—e.g., to accommodate more hosts or new policies—difficult to implement.

In such settings, Zeppelin helps by automating domain creation/restructuring. It can automatically find good routing domain hierarchies with increased resilience and policy compliance. We envision that operators will synthesize "one-shot" domain assignments for their changing input policies at coarse-grained timescales (days/weeks) when there are significant changes to their networks or policies; operators need not re-synthesize assignments for low-level policy changes.

In this subsection, we present an algorithm that searches the space of possible domain assignments Θ to find one that meets all configuration policies and increases connectivity-resilience by reducing the number of static routes. Note that a simple variant of this problem: finding a domain assignment with zero static routes is NP-complete (Theorem C.3.1). Therefore, we opt for a greedy stochastic search. Zeppelin uses Markov Chain Monte Carlo

Algorithm 6 Markov Chain Monte Carlo search for finding a domain assignment that minimizes the expression e

```
1: procedure MCMCSEARCH(e)
2:     Θ ← random domain assignment
3:     while max iterations OR timeout do
4:         γ = COST(Θ, e)
5:         Θ′ = RANDOMCHANGE(Θ)
6:         γ′ = COST(Θ, e)
7:         Set Θ = Θ′ with probability Pr(Θ → Θ′)
```

(MCMC) sampling methods, specifically the Metropolis-Hasting algorithm, a common technique used in optimization problems [78].

4.5.1 Searching Assignments with MCMC

MCMC sampling is a technique for drawing elements from a probability density function in direct proportion to its value. In our setting, MCMC searches the space of domain assignments and, if we assign higher probabilities to domain assignments with lower cost, MCMC will explore good configurations more *often* than bad ones. For MCMC to work we need to provide a transition function that lets us move from one domain assignment to another with a certain probability and a cost function that assigns costs (and therefore probabilities) to domain assignments.

In our setting, each domain assignment Θ has an associated cost $c(\Theta, e)$ where $e = expr(sc, bc)$ is the expression we are trying to minimize—e.g., if we are trying to minimize the number of static routes $e = sc$. The search starts by setting the current domain assignment to some random domain Θ_0. The following process then repeats. Given the current domain assignment Θ, compute a new domain assignment Θ' by randomly moving a gateway router r from one domain to another—i.e., $\Theta(r) \neq \Theta'(r)$ and for every $r' \neq r$, $\Theta(r') = \Theta'(r')$. If $c(\Theta', e) \leq c(\Theta, e)$, then Θ' becomes the current domain assignment. If $c(\Theta', e) > c(\Theta, e)$, then Θ' becomes the current domain assignment with probability $Pr(\Theta \rightarrow \Theta') = exp(-\beta \times (c(\Theta', e) - c(\Theta, e))$ (where β is a positive constant) while Θ continues being the current domain assignment with probability $1 - Pr(\Theta \rightarrow \Theta')$. The procedure is illustrated in Algorithm 6.

The algorithm always accepts a new proposal Θ' that has cost lower than Θ. If Θ' has a higher cost than Θ, the proposal is accepted with probability inversely proportional to how far the costs of Θ and Θ' are. This ensures that the algorithm does not get stuck at local minima, but explores proposals with small cost differences with higher probability.

4.5.2 The Cost of a Domain Assignment

We now look at how the cost $c(\Theta, e)$ is computed; this amounts to substituting in $e = expr(sc, bc)$ the values of sc and bc for the assignment Θ. The techniques presented in subsection 4.3.3 provide a way to synthesize BGP configurations and inter-domain static routes for a given domain assignment and can be used to efficiently compute the quantities bc. However, given a domain assignment, computing the minimal number of intra-domain static routes is hard. Since we want MCMC to explore as many assignments as possible, we present a heuristic technique for estimating the number of static routes.

Given two paths π and π' for destinations λ and λ', we define these paths form a diamond if these paths intersect at two routers (r_1 and r_2) without any common router in between. If the paths π and π' completely lie in the same domain, the presence of a diamond implies that there are two different shortest paths between r_1 and r_2, which means that at least one static route is required. On the other hand, if r_1 and r_2 lie in different domains, no static routes are required to resolve this diamond. Before starting the MCMC search, Zeppelin precomputes the set of all diamonds induced by the paths Π. For each domain assignment Θ, Zeppelin estimates the number of intra-domain static routes by counting the number of diamonds formed by every pair of paths which lie inside the same domain.

While diamonds in the same domain definitely require static routes, there might be sets of paths that do not contain diamonds but still require static routes; these are not taken into account in our estimate. However, our diamond-based estimate can be computed efficiently and our experiments show that reductions in cost lead to lesser static routes and increased connectivity-resilience (§4.6.3).

4.6 Evaluation

We implemented a prototype of Zeppelin in Python. For a given workload, Zeppelin outputs Quagga template configurations [72]. We use the Gurobi solver [46] for linear constraints generated by Zeppelin. In this subsection, we evaluate Zeppelin using enterprise-scale data center fat-tree topologies [8] of different sizes. Experiments were conducted on a 32-core Intel-Xeon 2.40GHz CPU machine and 128GB of RAM. Specifically, we ask the following questions.

Q1: How does Zeppelin perform on different workloads? (§4.6.1)
Q2: How resilient are Zeppelin's configurations? (§4.6.2)
Q3: Can Zeppelin get higher resilience by trying different domain assignments? (§4.6.3)
Q4: How does Zeppelin compare with the state-of-the-art? (§4.6.4)

4.6.1 Single Domain End-to-end Performance

We evaluate the end-to-end performance of Zeppelin on a fat-tree topology with 45 routers and a single OSPF domain. The size of the topology is consistent with operator preferences to restrict the size of a domain to under 50 routers. PC refers to the algorithm that handles complex policies and synthesizes policy-compliant configurations with few static routes (§4.3) and 1-WC (resp. 2-WC) refers to the algorithm that handles waypoint policies and can synthesize policy-compliant configurations with high policy-resilience from 1 path (resp. 2 paths) per waypoint policy (§4.4).

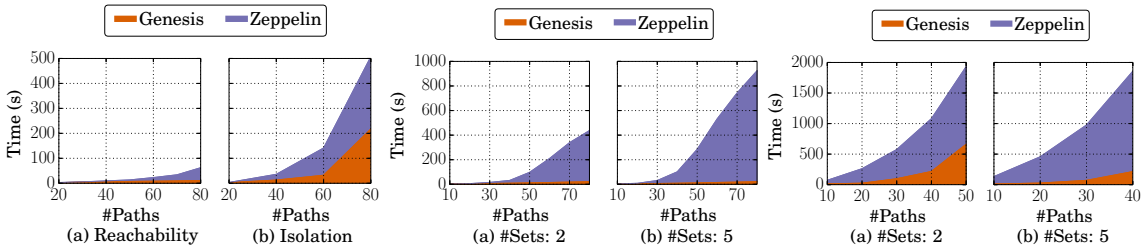
We consider two classes of policies. We evaluate algorithm PC to handle simple reachability policies and complex isolation policies that existing tools cannot handle; Our second class consists of waypoint policies of the form described in §4.4; for this class we evaluate 1-WC and 2-WC. For each algorithm we report the time for Genesis to generate policy-compliant forwarding paths and time Zeppelin takes to generate configurations from these paths. We set a timeout of 2,000 seconds for each workload.

Reachability Policies. We generate reachability policies (each of which corresponds to a path) with randomly generated endpoints. Figure 4.9(a)(a) shows the synthesis time for PC for varying number of paths. For these workloads, Zeppelin can synthesize configurations for 80 policies in 62 seconds, out of which configuration synthesis from paths takes 52 seconds on average.

Isolation Policies. We generate policies to ensure tenant-isolation in a multi-tenant topology, i.e., each tenant's traffic will be isolated from traffic from other tenants, and a tenant cannot interfere with any other tenant. This is achieved by adding isolation policies amongst different tenant's traffic. We have n tenant groups (between 1 to 4), each group comprised of g destinations (20). Figure 4.9(a)(b) shows the synthesis time for PC. The x-axis shows the paths $n * g$.

As the number of tenants increase, time to synthesize the data plane increases exponentially because finding isolated paths is NP-complete. Also, these paths are longer than if we only had reachability policies. Similarily, time taken to synthesize OSPF configurations increases exponentially with the number of tenants as PC needs to add more static routes and requires more iterations of the unsat-core learning procedure. For this workload, Zeppelin can synthesize configurations for 4 tenants, each with 20 destinations in 507 seconds, where Genesis takes 217 seconds for synthesizing paths. Note that synthesizing configurations is harder for isolation than reachability because of longer paths provided by Genesis.

Waypoint Policies. We generate waypoint policy workloads with varying number of destination subnets and waypoint sets as follows. We consider workloads with 2 and 5 unique waypoint sets where each set contains 3 randomly picked routers. Each waypoint set can be



(a) PC (b) 1-WC (c) 2-WC

Fig. 4.9 End-to-end synthesis time for waypoint policy workloads for varying number of paths and different number of waypoint sets.

viewed as a class of replicated middleboxes. We randomly generate policies that map each subnet to one of the waypoint sets.

Figure 4.9(b) and (c) show the end-to-end synthesis time for varying number of destination paths and the number of unique waypoint sets (2 and 5). Since synthesizing paths for waypoint policies can be done independently, Genesis synthesis time is small compared to OSPF synthesis. The synthesis times of 1-WC and 2-WC increase with the number of waypoint sets because the algorithms need to add constraints corresponding to $D(s,t,\mathbb{W})$ for each set. Hence, computing the unsat-core in each iteration to find static routes is more expensive. For 5 waypoint sets and 40 destinations, Zeppelin takes 120 seconds on average to synthesize waypoint-compliant configurations and 1800 seconds to synthesize policy-resilient configurations.

Q1: Zeppelin can synthesize policy-compliant configurations for medium size topologies in less than 10 minutes and policy-resilient configurations in less than an hour.

4.6.2 Resilience of Intra-domain Configurations

Connectivity-resilience. We measure the connectivity-resilience of the configurations generated using the algorithm PC on the reachability workload described in subsection 4.6.1. For our baseline, we use configurations where paths are enforced using only static routes and OSPF weights are assigned randomly. Figure 4.10 shows the results. A point above the diagonal denotes a benchmark in which Zeppelin generated a configuration with higher connectivity resilience than the baseline algorithm. PC provides an average connectivity-resilience score of 0.97 over the baseline score of 0.88. Moreover, PC places on average 9.3% of the static routes than required by the baseline.

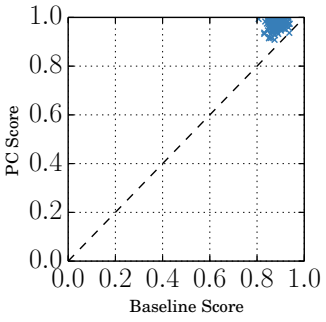
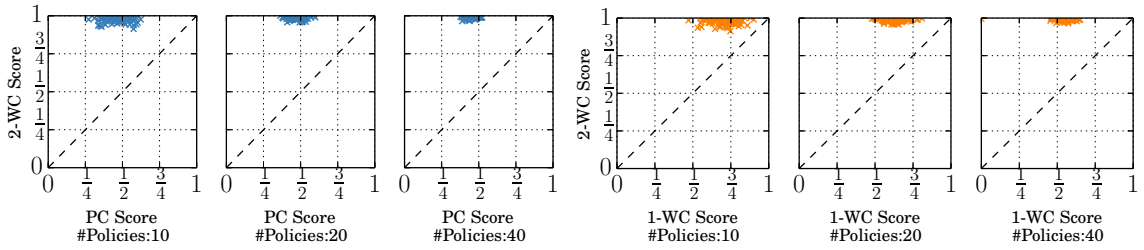


Fig. 4.10 Connectivity-resilience: PC vs baseline.



(a) 2-WC vs. PC (b) 2-WC vs. 1-WC

Fig. 4.11 Policy-resilience scores of 2-WC, 1-WC, and PC synthesis for varying waypoint workloads.

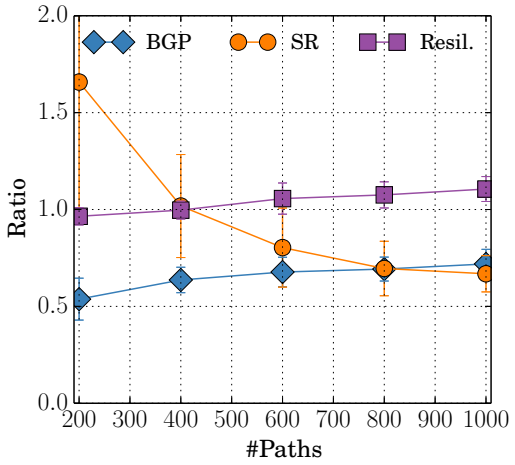


Fig. 4.12 MCMC Evaluation for varying number of paths.

Policy-resilience. We measure the policy-resilience of the configurations generated using algorithms 1-WC and 2-WC on the waypoint workload described in §4.6.1 with 5 waypoint sets. For each workload, we run each algorithm 3 times with a different random seed, and report the most-resilient configuration it obtains. For our baseline, we use the configurations generated by the algorithm PC, which only tries to reduce the number of static routes.

Figure 4.11 shows the results for varying number of policies (10, 20 and 40), each policy mapped to a particular destination IP address. For the plot on the left (resp. right) a point above the diagonal denotes a benchmark in which 2-WC generated a configuration with higher policy-resilience than PC (resp. 1-WC). For 40 policies, 2-WC can synthesize configurations with high policy-resilience (average 0.98) over PC (average 0.41) and 1-WC (average 0.51). Although not as good as 2-WC, 1-WC is able to provide higher policy-resilience than PC thanks to the relaxed constraints that do not require the path given by Genesis to be the shortest one.

Q2: Zeppelin can synthesize highly resilient configurations.

Topology	#Classes	SyNET	Zeppelin	Speedup
Internet2	1	9.0s	0.02s	**450**×
	5	21.3s	0.03s	**710**×
	10	49.3s	0.03s	**1,643**×
G3	1	9.4s	0.03s	**313**×
	5	19.8s	0.03s	**660**×
	10	39.9s	0.03s	**1,330**×

Fig. 4.13 Comparison of synthesis times of Zeppelin and SyNET for OSPF + Static route workloads.

4.6.3 Dynamic Domain Assignment Performance

In this subsection, we measure whether when Zeppelin is allowed to explore different domain assignments, the MCMC algorithm presented in §4.5 can generate configurations with higher resilience. We consider an 80 router fat-tree topology and run the MCMC sampling for 600s—i.e., more than 100,000 iterations—to minimize the cost function $max(sc, bc)$. Using this cost function, Zeppelin tries to jointly decrease number of BGP local preferences and static routes. For the input, we generate n (between 200 and 1,000) random paths for $n/4$ destination IPs, with random path length between 3 and 10. We require Zeppelin to split the network into 5 OSPF domains each with size in range between 4 and 10. We conduct each experiment 20 times and report averages and standard deviations. For each MCMC run, we collect the domain assignments with worst and best scores. For these assignments, we use the algorithm PC to compute path-compliant configurations. Figure 4.12 shows, for varying number of paths, the average ratios between the best- and worst-score configurations in terms of BGP configuration overhead, number of static routes, and connectivity-resilience. For smaller workloads, the algorithm reduces the BGP configuration overhead, but increases static routes, leading to worse connectivity-resilience. For larger workloads, the algorithm can reduce both static routes and BGP local preference entries by $0.3\times$ and improve the connectivity-resilience by $0.1\times$.

Q3: When allowed to try different domain assignments, **Zeppelin synthesizes configurations with higher connectivity-resilience mainly for workloads with large number of paths**.

4.6.4 Comparison to SyNET

SyNET [29] is a network-wide configuration synthesis system which supports multiple routing protocols (OSPF and BGP) and static routes. In this subsection, we compare SyNET's performance with Zeppelin for OSPF + static route configuration synthesis workloads obtained from the authors of SyNET.

For these experiments, we use the two network topologies the SyNET's authors made available: (1) Internet2: a US-based network with 9 routers, and (2) G3: A 3×3 grid topology. The routing requirements define the forwarding behavior for 1, 5, and 10 traffic classes as follows: for each router and traffic class the workload specifies the next-hop router and the protocol for forwarding: OSPF or static route. Thus, for a topology with n routers and m traffic classes, the workload contains $n \times m$ SyNET requirements. We combine these requirements to generate paths to provide as input for Zeppelin's OSPF synthesis. Since the workload specifies which routers had static routes installed for the different classes, we add these static routes (and remove the corresponding constraints), and use the algorithm presented in subsection 4.3.2.1 to find the OSPF weights.

Figure 4.13 shows Zeppelin outperforms SyNET for all the workloads and topologies, achieving a best case 1,643$\times$ speedup for Internet2 topology with 10 traffic classes. In each of these experiments, Zeppelin invokes the LP solver once and does not need to add more static routes than specified in the input requirements, which matches with the SyNET output (if more static routes were required, SyNET would have returned no solution for these experiments).

Q4: Zeppelin is able to achieve 2-3 orders of magnitude speedup over state-of-the-art tools.

4.7 Related Work

Centralized control. RCP [20] supported logically central BGP configuration. The more recent Fibbing [89] system provides centralized control over distributed routing by creating fake nodes and fake links to steer the traffic in the network through paths that are not the shortest, similar to how Zeppelin uses static routes. Both approaches face the issue of forwarding loops in the network during failures. Fibbing and Zeppelin differ in the specific algorithms to solve the synthesis problem. Zeppelin does offer key advantages: first, by using static routes for steering, we do not increase the control traffic in the network, unlike the fake advertisements used in Fibbing. Second, Fibbing by design targets a single domain and does not take into account domain decomposition and inter-domain routing.

Configuration synthesis. ConfigAssure [66] uses a combination of logic programming and SAT solving to synthesize network configurations for security, functionality, performance and reliability requirements specified as constraints; but it does not support any notion of policy- or connectivity-resilience or hierarchical domain splitting. Fortz et al. [33] tackle the problem of optimizing OSPF weights for performing traffic engineering, but their work is tailor-made to just this specific problem. Propane [11, 12] tackles the specific problem of synthesizing

BGP configurations for concrete and abstract topologies to ensure network-wide objectives hold even under failures. Propane is suited to specify preferences on paths and peering policies among different autonomous systems. Propane translates policies to a graph-based intermediate representation, which is then compiled to device-level BGP configurations. The automata-based compilation and resilience algorithms for Propane are tailored for its underlying technology (BGP configurations with support for local preferences, MEDs and communities), in contrast to our LP-based algorithms which synthesize hierarchical BGP, OSPF and static routing configurations which have a different forwarding model than BGP.

SyNET [29] tackles network-wide configuration synthesis (Definition 3) by modeling the behavior and interactions of the routing protocols as a stratified Datalog program, and using SMT to synthesize the Datalog input such that the fixed point of the Datalog program (which represents the network's forwarding state after convergence) satisfies certain policies or path requirements. While both systems can take paths as input requirements, Zeppelin uses a two-phase approach where OSPF weights are synthesized separately using LP-solvers—which are faster and parallelizable—rather than directly solving the whole configuration synthesis problem using SMT solvers. Moreover, SyNET's approach does not deal with resilience, a key aspect we tackle in this work, and SyNET does not attempt to minimize the number of static routes, which can cause undesirable behaviors like routing loops. Finally, SyNET supports routers that run both OSPF and BGP protocols and can be configured with static routes, which does not fit naturally into a hierarchical structure where some routers only run OSPF and not BGP. In contrast, our stochastic MCMC algorithm can look for dynamic domain assignments with constraints on OSPF domain sizes, lower BGP configuration complexity and higher resilience.

Policy languages. Zeppelin uses the Genesis [86] framework to synthesize paths which comply with reachability, waypoint and isolation policies. Genesis is tailored for software-defined networks comprised of OpenFlow [63] switches, whose forwarding tables can be directly programmed to specify the next-hop switch for different classes of traffic based on the paths provided by Genesis. Zeppelin's algorithms cater for legacy control planes running OSPF and BGP which lack the programmability of SDNs. Genesis also provides resilience in a different manner than Zeppelin. Genesis precomputes and installs backup paths (generated using isolation policies) in the SDN dataplane, while Zeppelin synthesizes OSPF weights and static routes such that, the distributed protocols converge to a policy-compliant path even under failures. Unlike Genesis, which can provide resilience for arbitrary k-link failures, Zeppelin currently supports only $k = 1$ link failure scenarios.

In the future, Zeppelin could use other policy language frameworks as a front-end (with less rich policy support but better performance). In Merlin [83], data planes that adhere to

policies expressed using regular expressions and min and max bandwidth guarantees are synthesized using mixed integer linear programming (ILP). NetKAT [9] is a domain-specific language and logic for specifying and verifying network packet-processing functions for SDN, based on Kleene algebra with tests (KAT). NetKAT can express certain network-wide policies like reachability and waypoints. However, both Merlin and NetKAT do not support link-isolation to produce edge-disjoint paths, which is needed for waypoint-compliance in §4.4.

Chapter 5

Conclusion and Future Work

In this thesis, we designed and evaluated three systems which are furthering the vision of intent-based networking. In this closing chapter, we summarize our key contributions and present directions for future research.

5.1 QARC

In Chapter 2, we presented QARC, a control plane abstraction to support verification of network link overload under different failure scenarios. QARC can efficiently verify complex network configurations thanks to its new Mixed-Integer-Programming encoding. QARC can also be used for determining which link capacities need to be upgraded to prevent load violations. We use QARC to show network load violations can occur in existing datacenter and ISP networks. QARC used the ARC abstraction which does not support complex BGP constructs like local preferences and communities. For future work, we would like to extend QARC to support more protocol constructs for generality. QARC can be extended to support verification of other quantitative properties like latencies and rate-limits. Finally, to tackle general repair for quantitative properties, joint changes to the network control plane and link capacities are required.

5.2 Genesis

In Chapter 3, we presented Genesis, a general and extensible network management system for multi-tenant datacenter networks. It leverages fast SMT solvers to synthesize data plane configuration from high level policies. Genesis incorporates novel ideas to significantly speed up synthesis, leveraging the hierarchical nature of datacenter network topologies and

the structure of the interaction between tenants' policies. Fine-grained traffic engineering based on online demand/flow size estimation and rapid rerouting is also crucial for datacenter workloads, and extending Genesis's TE policies to fine-grained timescales is subject of future work. Also, the performance of SMT solvers with optimization objectives is quite slow, and calls for domain-specific techniques to speed up the synthesis. Also, datacenter networks are highly symmetrical, and this symmetry can be leveraged to speed up synthesis (similar to the work of Plotkin et al. [69] to speed up network verification using symmetry). The main challenges of using symmetry in synthesis is considering two aspects of symmetry: network symmetry and policy symmetry. Also, our treatment of resilience synthesis is preliminary and future work will be geared towards synthesizing resilient forwarding planes incorporating capacity constraints and traffic engineering.

5.3 Zeppelin

In Chapter 4, we presented Zeppelin, a system for automatically synthesizing highly-resilient distributed hierarchical control planes—i.e., OSPF and BGP router configurations—from high-level policies. For hierarchical control planes, we devised algorithms based on our OSPF-BGP interaction model. In practice, there exists other hierarchical models, and extending Zeppelin's modular two-phase approach to synthesize these routing models is subject for future work. Another direction for future work is to modify OSPF synthesis to incorporate multi-path routing for load-balancing etc. Finally, resilience is an important requirement, and extending Zeppelin to support other notions of policy-resilience for $k > 1$ link failures is subject for future work.

References

[1] Cisco IOS configuration fundamentals command reference. http://cisco.com/c/en/us/td/docs/ios/fundamentals/command/reference/cf_book.html.

[2] Floodlight sdn controller. http://www.projectfloodlight.org/floodlight/.

[3] Intent: Don't tell me what to do! (tell me what you want). https://www.sdxcentral.com/articles/contributed/network-intent-summit-perspective-david-lenrow/2015/02/.

[4] Python lex-yacc. http://www.dabeaz.com/ply/.

[5] Batfish. https://github.com/batfish/batfish, 2018.

[6] ARC. http://bitbucket.org/uw-madison-networking-research/arc, 2019.

[7] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, Santa Clara, CA, Feb. 2020. USENIX Association.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[9] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[10] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 155–168, New York, NY, USA, 2017. ACM.

[11] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the ACM SIGCOMM 2016 Conference on SIGCOMM*, SIGCOMM '16, 2016.

[12] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 437–451. ACM, 2017.

[13] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[14] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[15] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[16] T. Benson, A. Akella, and A. Shaikh. Demystifying configuration challenges and trade-offs in network-based isp services. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 302–313, New York, NY, USA, 2011. ACM.

[17] T. Benson, A. Akella, and A. Shaikh. Demystifying configuration challenges and trade-offs in network-based isp services. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 302–313, New York, NY, USA, 2011. ACM.

[18] N. Bjorner and A.-D. Phan. νz - maximal satisfaction with z3. In T. Kutsia and A. Voronkov, editors, *SCSS 2014. 6th International Symposium on Symbolic Computation in Software Science*, volume 30 of *EPiC Series in Computer Science*, pages 1–9. EasyChair, 2014.

[19] P. Broström and K. Holmberg. Compatible weights and valid cycles in non-spanning ospf routing patterns. *Algorithmic Operations Research*, 4(1):19–35, 2009.

[20] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association.

[21] Y. Chang, S. Rao, and M. Tawarmalani. Robust validation of network designs under uncertain demands and failures. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 347–362, Boston, MA, 2017. USENIX Association.

[22] M. Chiesa, A. Gurtov, A. Madry, S. Mitrovic, I. Nikolaevskiy, M. Shapira, and S. Shenker. On the Resiliency of Randomized Routing Against Multiple Edge Failures. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 134:1–134:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[23] J. W. Chinneck. *Feasibility and Infeasibility in Optimization:: Algorithms and Computational Methods*, volume 118. Springer Science & Business Media, 2007.

[24] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *J. Artif. Int. Res.*, 40(1):701–728, Jan. 2011.

[25] Cisco. What is administrative distance. http://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/15986-admin-distance.html, 2013.

[26] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[27] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] V. Diekert and P. Gastin. First-order definable languages. In *Logic and Automata: History and Perspectives, Texts in Logic and Games*, pages 261–306. Amsterdam University Press, 2008.

[29] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. Network-wide configuration synthesis. In *29th International Conference on Computer Aided Verification, Heidelberg, Germany, 2017*, CAV'17, 2017.

[30] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, 2018. USENIX Association.

[31] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 217–232. USENIX Association, 2016.

[32] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, 2015.

[33] B. Fortz and M. Thorup. Internet traffic engineering by optimizing ospf weights. In *INFOCOM 2000. Nineteenth annual joint conference of the IEEE computer and communications societies. Proceedings. IEEE*, volume 2, pages 519–528. IEEE, 2000.

[34] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[35] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva. Probabilistic netkat. In *European Symposium on Programming Languages and Systems*, pages 282–309. Springer, 2016.

[36] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 359–373. ACM, 2017.

[37] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the ACM SIGCOMM 2016 Conference on SIGCOMM*, SIGCOMM '16, 2016.

[38] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. Technical report, University of Wisconsin-Madison, 2016.

[39] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[40] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 395–408, New York, NY, USA, 2015. ACM.

[41] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, IMC '15, pages 395–408, New York, NY, USA, 2015. ACM.

[42] N. Giannarakis, D. Loehr, R. Beckett, and D. Walker. Nv: An intermediate language for verification of network control planes. In *PLDI*. ACM, 2020.

[43] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, New York, NY, USA, 2011. ACM.

[44] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[45] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[46] Gurobi. Gurobi Optimization. http://www.gurobi.com/, 2017.

[47] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan. Measuring control plane latency in sdn-enabled switches. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 25:1–25:6, New York, NY, USA, 2015. ACM.

[48] V. Heorhiadi, M. K. Reiter, and V. Sekar. Simplifying software-defined network optimization using sol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 223–237, 2016.

[49] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 15–26, New York, NY, USA, 2013. ACM.

[50] C. Hopps. Analysis of an equal-cost multi-path algorithm, 2000.

[51] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 539–550, New York, NY, USA, 2014. ACM.

[52] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 13–24, New York, NY, USA, 2013. ACM.

[53] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[54] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[55] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 1–14, New York, NY, USA, 2015. ACM.

[56] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, P. Lapukhov, C. L. Lim, and R. Soulé. Semi-oblivious traffic engineering: The road not taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 157–170, Renton, WA, 2018. USENIX Association.

[57] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 527–538, New York, NY, USA, 2014. ACM.

[58] G. S. Malkin. Rip version 2. STD 56, RFC Editor, November 1998. http://www.rfc-editor.org/rfc/rfc2453.txt.

[59] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjálmtýsson, and A. Greenberg. Routing design in operational networks: A look from the inside. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 27–40, New York, NY, USA, 2004. ACM.

[60] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjálmtýsson, and A. Greenberg. Routing design in operational networks: A look from the inside. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 27–40, New York, NY, USA, 2004. ACM.

[61] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM Transactions on Networking (TON)*, 16(4):749–762, 2008.

[62] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient synthesis of network updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 196–207, New York, NY, USA, 2015. ACM.

[63] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[64] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[65] J. Moy. Ospf version 2. STD 54, RFC Editor, April 1998. http://www.rfc-editor.org/rfc/rfc2328.txt.

[66] S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Netw. Syst. Manage.*, 16(3):235–258, Sept. 2008.

[67] O. Padon, N. Immerman, A. Karbyshev, O. Lahav, M. Sagiv, and S. Shoham. Decentralizing sdn policies. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 663–676, New York, NY, USA, 2015. ACM.

[68] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.

[69] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 69–83, New York, NY, USA, 2016. ACM.

[70] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 29–42, New York, NY, USA, 2015. ACM.

[71] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38, New York, NY, USA, 2013. ACM.

[72] Quagga. Quagga Routing Suite. http://www.nongnu.org/quagga/, 2017.

[73] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 109–114. ACM, 2013.

[74] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). Technical report, 2005.

[75] M. Roughan, A. Greenberg, C. Kalmanek, M. Rumsewicz, J. Yates, and Y. Zhang. Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, pages 91–92, New York, NY, USA, 2002. ACM.

[76] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[77] S. Saha, S. Prabhu, and P. Madhusudan. Netgen: Synthesizing data-plane configurations for network policies. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 17:1–17:6, New York, NY, USA, 2015. ACM.

[78] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 305–316, New York, NY, USA, 2013. ACM.

[79] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[80] R. Singh, M. Ghobadi, K.-T. Foerster, M. Filer, and P. Gill. Radwan: Rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 547–560, New York, NY, USA, 2018. ACM.

[81] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha. A fast compiler for netkat. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 328–341, New York, NY, USA, 2015. ACM.

[82] S. Smolka, P. Kumar, N. Foster, D. Kozen, and A. Silva. Cantor meets scott: Semantic foundations for probabilistic networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 557–571, New York, NY, USA, 2017. ACM.

[83] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 213–226, New York, NY, USA, 2014. ACM.

[84] B. Stephens, A. L. Cox, and S. Rixner. Plinko: Building provably resilient forwarding tables. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 26:1–26:7, New York, NY, USA, 2013. ACM.

[85] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella. Detecting network load violations for distributed control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 974–988, New York, NY, USA, 2020. Association for Computing Machinery.

[86] K. Subramanian, L. D'Antoni, and A. Akella. Genesis: Synthesizing forwarding tables for multi-tenant networks. In *POPL*. ACM, 2017.

[87] K. Subramanian, L. D'Antoni, and A. Akella. Synthesis of fault-tolerant distributed router configurations. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(1):22:1–22:26, Apr. 2018.

[88] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439. ACM, 2016.

[89] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central control over distributed routing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 43–56, New York, NY, USA, 2015. ACM.

[90] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott. Fsr: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Trans. Netw.*, 20(6):1814–1827, Dec. 2012.

[91] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg. Cope: Traffic engineering in dynamic networks. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 99–110, New York, NY, USA, 2006. ACM.

[92] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Scalable verification of border gateway protocol configurations with an smt solver. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 765–780, New York, NY, USA, 2016. ACM.

[93] Y. Yuan, R. Alur, and B. T. Loo. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 20:1–20:7, New York, NY, USA, 2014. ACM.

[94] S. Zhang, F. Ivancic, C. Lumezanu, Y. Yuan, A. Gupta, and S. Malik. An adaptable rule placement for software-defined networks. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 88–99, June 2014.

[95] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 5:1–5:14, New York, NY, USA, 2014. ACM.

[96] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 73–85, Berkeley, CA, USA, 2015. USENIX Association.

[97] D. Zhuo, M. Ghobadi, R. Mahajan, K.-T. Förster, A. Krishnamurthy, and T. Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 362–375, New York, NY, USA, 2017. ACM.

Appendix A

QARC Proofs

A.1 Verification Proofs

Lemma 1. *For every traffic class $tc \in TC$ every failure scenario $FL \in \overline{FL}$, Constraints 2.1-2.3 ensure that (1) the Flow Graph $FG(tc, FL)$ is a directed acyclic graph and (2) flow is preserved from $Src(tc)$ to $Dst(tc)$.*

Proof. We proceed by contradiction. First, let us assume that a router r drops some amount of traffic, i.e., the incoming flow into r is not equal to outgoing flow of r. However, this contradicts Constraint 2.2, thus, (2) flow is conserved at every router.

Second, let us assume that the flow graph contains a cycle containing router r and flow amount f_c which keeps flowing in the cycle. The flow f_c will not reach $Dst(tc)$, and since, flow is conserved at every router, the flow reached at destination will not contain f_c. However, this contradicts Constraint 2.1 that all flow from source is reached at the destination. Thus, proved that (1) the flow graph will not contain any cycles. □

Lemma 2. *For every traffic class $tc \in TC$ and failure scenario $FL \in \overline{FL}$, Constraints 2.1-2.7 ensure that each path from Src(tc) to Dst(tc) in the Flow Graph $FG(tc, FL)$ is a shortest weighted path in $ETG(tc)$ under the failure scenario FL.*

Proof. We proceed by contradiction. Let us assume there is a path p in Flow Graph $FG(tc, FL)$ which is not a shortest path in $ETG(tc)$, i.e., there is a shortest path p' and $W(p') < W(p)$ (where $W(p)$ denotes the sum of weights of edges in path p). We denote the flow on path p as f_p and flow on path p' as $f_{p'}$.

By virtue of Constraint 2.6, we can write $Dst(src(tc)) = W(p) * f_p + W(p') * f_{p'} + ...$ Using Lemma 1, the sum of flows on all paths must be equal to 1 and p' is a shortest path, therefore, $Dst(src(tc)) > W(p')$.

Consider Constraint (2.5) for routers on the path $p' = Src(tc) \rightarrow r_1 \ldots Dst(tc)$. p' is in the flow graph, so no edge will be failed and Fail variables on p' will be set to 0.

$$\begin{aligned} Dist(Src(tc),tc) &\leq W((Src(tc),r_1)) + Dist(r_1,tc) \\ &\leq W((Src(tc),r_1)) + W((r_1,r_2) + Dist(r_2,tc) \ldots \\ &\leq W(p') + Dist(Dst(tc),tc) \end{aligned}$$

Since, $Dist(Dst(tc),tc) = 0$, $Dist(Src(tc),tc) \leq W(p')$. This contradicts our previous assertion that $Dst(src(tc)) > W(p')$. Hence, proved. □

Theorem 2.3.1. *For every traffic class $tc \in TC$ and failure scenario $FL \in \overline{FL}$, Constraints 2.1-2.11 ensure that the flow Graph $FG(tc,FL)$ is a directed acyclic graph such that each path from $Src(tc)$ to $Dst(tc)$ is the shortest path in $ETG(tc)$, and for every router n in flow graph $FG(tc,FL)$, the flows on outgoing edges which lie on shortest paths from n to $Dst(tc)$ are equal., i.e., $\forall n_1, n_2 \in next(n).\ F_{tc}((n,n_1)) = F_{tc}((n,n_2))$.*

Proof. Lemma 1 and Lemma 2 prove that the flow graph $FG(tc,FL)$ does not contain any cycles and only contains the shortest paths in $ETG(tc)$. To prove that for any router n, flow on outgoing edges which lie on shortest paths are equal, we proceed by contradiction. Let us assume that for traffic class tc, router n does not split traffic equally, i.e., $\exists n_1, n_2 . F_{tc}((n,n_1)) \neq F_{tc}((n,n_2))$. In terms of the QARC variables, the assumption translates to $Flow((n,n_1),tc) \neq Flow((n,n_2),tc)$. Without loss of generality, let $Flow((n,n_1),tc) < Flow((n,n_2),tc)$.

By virtue of Constraints 2.8, the following holds:

$$Flow((n,n_1),tc) < Flow((n,n_2),tc) \leq maxFlow(n,tc) \tag{A.1}$$

By Lemma 2, both n_1 and n_2 lie on shortest paths from n to the destination. Therefore, the expression $W((n,n_1)) + Dist(n_1,tc) - Dist(n,tc) + Fail((n,n_1))$ is equal to 0 (similarly for n_2). Plugging in Constraints (2.9), we get:

$$minFlow(n,tc) \leq Flow((n,n_1),tc) < Flow((n,n_2),tc) \tag{A.2}$$

From (A.1) and (A.2), we get

$$minFlow(n,tc) < maxFlow(n,tc)$$

However, the above assertion contradicts Constraint (2.10). Hence, proved. □

Theorem 2.4.1 (Correctness)**.** *A failure scenario FL satisfies Constraints 2.1-2.11, 2.13-2.14 if and only if there exists a link $l \in Links$ such that $Util_{FL}(l) \geq Cap(l)$.*

Proof. First, we prove that if FL satisfies Constraints 2.1-2.11, 2.13-2.14, then one of the links' utilization will exceed capacity. We proceed by contradiction. Let us assume that there exists FL such that for all $l \in Links$, $Util_{FL}(l) < Cap(l)$, i.e., no link's utilization exceeds capacity. In Constraint 2.13, the numerator in the fractional term in the LHS represents the amount of traffic on the link: $Util_{FL}(l)$. Therefore, for all links:

$$Load(l) - Util_{FL}(l) > Cap(l) \leq 0 Load(l) < 1 \tag{A.3}$$

However, this contradicts with Constraint (2.14). Hence, proved.

We now prove if there exists a FL such that there is a link $l \in Links$ where $Util_{FL}(l) \geq Cap(l)$, then there is a satisfying assignment for Constraints 2.1-2.11 2.13-2.14. Using Theorem 2.3.1 and Theorem 2.2.1, the flow graphs in the actual network will be a satisfying assignment for Constraints 2.1-2.11.

For link l, we set $Load(l) = 1$ and for all other links $l' \in Links \setminus l$, we set $Load(l') = 0$. Since $Util_{FL}(l) \geq Cap(l)$:

$$\frac{Util_{FL}(l)}{Cap(l)} - Load(l) = \frac{Util_{FL}(l)}{Cap(l)} - 1 \geq 0$$

Thus, Constraint 2.13 is satisfied for link l. For other links $l' \in Links \setminus l$, $\frac{Util_{FL}(l)}{Cap(l)} > 0$ as both quantities are positive, thus Constraint 2.13 is satisfied for all links. Finally, since $Load(l) = 1$, Constraint 2.14 is satisfied as well. Hence, proved. □

A.2 Minimization Proofs

Theorem A.2.1 (Soundness)**.** *If edge e in $ETG(tc)$ has been removed in the minimized $ETG_{min}(tc)$, then for all failure scenarios $FL \in \overline{FL}$, e will not in be the flow graph $FG(tc, FL)$.*

Proof. We proceed by contradiction. Let us assume there exists a failure scenario $FL \in \overline{FL}$ such that $e = (n_1, n_2)$ is not in $ETG_{min}(tc)$ but is present in $FG(tc, FL)$.

Using Lemma 2, since e is present in $FG(tc, FL)$, it is on a shortest path from $Src(tc)$ to $Dst(tc)$ in $ETG(tc)$. Therefore, $SP(Src(tc), n_1) + W(n_1, n_2) + SP(n_2, Dst(tc)) = SP(Src(tc), Dst(tc))$.

Since e is not in the minimized ETG, Condition (2.16) holds, therefore:

$$SP(Src(tc),n_1)+W(n_1,n_2)+SP(n_2,Dst(tc)) > MaxDist$$
$$SP(Src(tc),Dst(tc)) > MaxDist \tag{A.4}$$

However, $maxDist = MAX_{FL\in\overline{FL}}Dist(Src(tc),tc)$, thus, $maxDist \geq SP(Src(tc),Dst(tc))$. This contradicts our assertion (A.4). Hence, proved. □

Theorem A.2.2 (Completeness)**.** *If edge e in $ETG(tc)$ has not been removed in the minimized $ETG_{min}(tc)$, then $\exists FL \in \overline{FL}$ such that e will be in the flow graph $FG(tc,FL)$.*

Proof. We proceed by contradiction. Assume that is $\forall FL \in \overline{FL}$, $e = (n_1,n_2)$ is not in $FG(tc,FL)$.

Since e was not removed by ETG minimization, using Condition (2.16):

$$SP(Src(tc),n_1)+W(n_1,n_2)+SP(n_2,Dst(tc)) \leq MaxDist$$

From Objective (2.15), $\exists FL$ such that $SP(Src(tc),n_1)+W(n_1,n_2)+SP(n_2,Dst(tc)) \leq Dist(Src(tc))$, which implies e lies on a shortest path from source to destination. Using Lemma 2, if e is on a shortest path from $Src(tc)$ to $Dst(tc)$, then e is in $FG(tc,FL)$. This contradicts our assumption that e in not in the flow graphs under all failure scenarios in $\overline{FL}$. Hence, proved. □

A.3 Upgrade Proofs

Theorem A.3.1. *If there is link l such that $Cap(l)$ is strictly less than computed capacity $Cap'(l)$, then there exists a failure scenario $FL \in \overline{FL}$ such that $Util_{FL}(l) > Cap(l)$.*

Proof. We proceed by contradiction. Let us assume there is no failure scenario $FL \in \overline{FL}$ such that $Util_{FL}(l) > Cap(l)$. Therefore, the following holds.

$$\max_{FL\in\overline{FL}} Util_{FL}(l) \leq Cap(l) \tag{A.5}$$

Objective 2.17 is used to compute the new capacity $Cap'(l)$:

$$Cap'(l) = \max_{FL\in\overline{FL}} \sum_{tc\in TC} [Flow(l,tc)+\Delta(l,tc)] \times T(tc)$$
$$Cap'(l) = \max_{FL\in\overline{FL}} Util_{FL}(l)$$
$$Cap'(l) \leq Cap(l) \quad \text{[From (A.5)]}$$

However, this contradicts our assertion that $Cap(l) < Cap'(l)$. Hence, proved. □

Appendix B

Genesis Proofs

B.1 NP-completeness Proof of Enforcing Isolation Policies

Theorem B.1.1. *Enforcing reachability and isolation policies is NP-complete.*

Proof. We show that the graph 3-coloring problem, which is NP-complete reduces to the enforcement problem for reachability and isolation policies. The latter is also in NP, so after the reduction we can conclude that it is also NP-complete.

Let $G = (V,E)$ be an instance of the graph 3-coloring problem. The graph G admits a 3-coloring if there exists a function $f : G \mapsto \{R,G,B\}$ such that for every edge $(v1,v2) \in E$, $f(v_1) \neq f(v_2)$.

We now show how to construct a topology $T = (S,L)$ and a corresponding set of policies P that can be enforced iff G admits 3-coloring. The topology T is the one depicted in Figure B.1. The set of flows is V, $v \in V$ we add a reachability policy $s >> d$ on the flow v. For each edge $(v1,v2) \in E$ we add an isolation policy $v_1||v_2$.

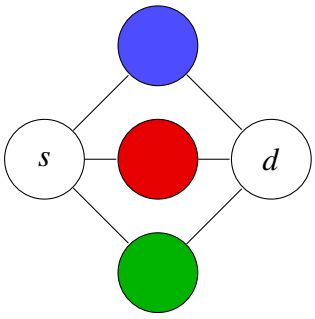


Fig. B.1 The switch topology T. All circles represent switches and all reachability policies are s to d

We now prove that if the policies can enforced then G admits 3-coloring. If the policy is enforced, we have a path from s to d for each flow/vertex v. We can set $f(v)$ to the color of the

corresponding middle-switch being crossed. Since for every $(v1, v2) \in E$ the two flows for v_1 and v_2 cannot share any edge, they must have crossed two middle-switches with different colors and f is a correct 3-coloring. The other direction of the proof is analogous. □

B.2 Resilience Transformation Proofs

Theorem 3.5.1 (Soundness)**.** *Given input policies* (PC, I)*, the data plane* $\hat{\xi}_{pc}$ *for every packet class* $pc \in PC$ *synthesized from transformed policies* (PC^R, I^R) *is* t-resilient *and policy-compliant.*

Proof. Assume, $\exists pc$ such that the data plane $\hat{\xi} = (S, L_{pc})$ is not *t-resilient*. Therefore, there exists a failure scenario θ such that $|\theta| \leq t$ and $\hat{\xi}_\theta = (S, L_{pc} \setminus \theta)$ is not resilient, i.e., there is no path from the source to destination. Thus, θ disabled all the paths of $\hat{pc}$.
However, the paths are edge-disjoint as each class in $\hat{pc}$ has a link-isolation policy with each other. Thus, t link failures cannot affect $t+1$ edge-disjoint paths of $\hat{pc}$. Thus, θ disabling all paths of $\hat{pc}$ is a contradiction.
Let us consider policy-compliance. Given a failure scenario $\theta \in \Theta(t)$, each data plane ξ of class pc has an active path. Consider a isolation policy in I: $pc_1 || \ pc_2$. In line 13 of Algorithm 1, each class of $\hat{pc_1}$ will be isolated to each class of $\hat{pc_2}$, thus any path of the data planes of pc_1 and pc_2 will satisfy the input policy $pc_1 || \ pc_2$. Hence, the data planes are policy-compliant. □

Theorem 3.5.2 (Completeness)**.** *Given input policies* (PC, I) *such that* $I = \emptyset$*, the synthesized data plane* ξ *for a packet class* pc *is* t-resilient *if and only if it contains* $t+1$ *edge-disjoint paths from source to destination for* pc*.*

Proof. Let us assume the data-plane $\hat{\xi}$ for pc is *t-resilient* such that less than $t+1$ instances of pc are required for resilience i.e. $|\hat{pc}| < t+1$. Let us consider a t-link failure scenario $\theta, |\theta| = t$ where a link is picked from each path of $\hat{pc}$. The active data plane $\hat{\xi}_\theta = (S, L_{pc} \setminus \theta)$ will not contain a path from source to destination as all paths of $\hat{pc}$ would be disabled. This is a contradiction since $\hat{\xi}$ is *t-resilient*. Therefore, we need atleast $t+1$ instances of pc to ensure *t-resilience*.

Now let us assume there are $t+1$ paths in a *t-resilient* $\hat{\xi}$ for pc, and not all of them are edge-disjoint. Consider two paths π_1 and π_2 in $\hat{\xi}$ which share a link. We can choose the shared link of π_1 and π_2 and $t-1$ links from the other $t-1$ paths and construct a failure scenario θ, $|\theta| = t$. The active data plane $\hat{\xi}_\theta = (S, L_{pc} \setminus \theta)$ will not contain a path from source to destination as all paths of $\hat{pc}$ would be disabled. This is a contradiction since ξ is *t-resilient*. Therefore, the $t+1$ paths of pc must be edge-disjoint.

Thus, the resilience transformation is sound and complete for a packet class if there are no isolation policies. □

B.3 Tactics Proofs

Theorem 3.6.1 (Soundness)**.** *For a tactic set* Γ*, if* $(Fwd, Reach) \models \Psi \wedge \bigwedge_{(R,pc)\in\Gamma} \Psi_T(R, pc))$*, then* $\forall (R, pc) \in \Gamma.\ \forall (\pi', pc') \in \Pi.\ pc = pc' \implies \Phi(\pi') \in L(R)$.

Proof. **Assume** $\exists (R, pc) \in \Gamma$ such that $(\pi, pc) \in \Pi$ and $\Phi(\pi) \notin L(R)$. Consider the three types of tactics for R:
Type 1: $R = \neg(l_{src}.^{i}.^{*}l_{dst})$
$\Phi(\pi) \notin L(R) \implies \Phi(\pi) \in L(l_{src}.^{i}.^{*}l_{dst})$.
Thus, $\pi = src\ sw_1 \ldots sw_i \ldots dst$.
Therefore, $(dst, pc, k_{dst}) \in Reach, k_{dst} \geq i+1$.
However, $(Fwd, Reach) \models \Psi_T(R, pc) \implies (Fwd, Reach) \models \forall sw, k \geq i+1.(sw, pc, k) \notin Reach$.
Contradiction, as $\exists sw, k \geq i+1.(sw, pc, k) \in Reach$.

Type 2: $R = \neg(l_{src}.^{i}\ l\ .^{*}l_{dst})$
$\Phi(\pi) \notin L(R) \implies \Phi(\pi) \in L(l_{src}.^{i}\ l\ .^{*}l_{dst})$.
Thus, $\pi = src\ sw_1 \ldots sw_i\ sw_{i+1} \ldots dst$ such that $\phi(sw_{i+1}) = l$. Also $sw_{i+1} \neq dst$ (dst has to be the last switch in π)
Therefore, $(sw_{i+1}, pc, i+1) \in Reach$.
However, $(Fwd, Reach) \models \Psi_T(R, pc) \implies (Fwd, Reach) \models \forall sw.\ \phi(sw) = l \wedge sw \neq dst \implies (sw, pc, i+1) \notin Reach$.
Contradiction, as $\exists sw.\ \phi(sw) = l\ \wedge\ sw \neq dst \wedge (sw, pc, i+1) \in Reach$.

Type 3: $R = \neg(l_{src}.^{i}\ l_1\ l_2\ .^{*}l_{dst})$
$\Phi(\pi) \notin L(R) \implies \Phi(\pi) \in L(l_{src}.^{i}\ l_1\ l_2\ .^{*}l_{dst})$.
Thus, $\pi = src\ sw_1 \ldots sw_i\ sw_{i+1}\ sw_{i+2} \ldots dst$ such that
$\phi(sw_{i+1}) = l_1, \phi(sw_{i+2}) = l_2$. Also $sw_{i+2} \neq dst$ (dst has to be the last switch in π)
Therefore, $(sw_{i+1}, pc, i+1) \in Reach \wedge (sw_{i+1}, sw_{i+2}, pc) \in Fwd$.
However, $(Fwd, Reach) \models \Psi_T(R, pc) \implies (Fwd, Reach) \models \forall n_1, n_2.\ \phi(n_1) = l_1\ \wedge\ \phi(n_2) = l_2\ \wedge\ n_2 \neq dst \implies \neg(Reach(n_1, pc, i+1) \wedge Fwd(n_1, n_2, pc))$.
Contradiction, as $\exists n_1.n_2.\ \phi(n_1) = l_1\ \wedge\ \phi(n_2) = l_2\ \wedge\ n_2 \neq dst \wedge Reach(n_1, pc, i+1) \wedge Fwd(n_1, n_2, pc)$. □

Theorem 3.6.2 (Completeness)**.** *For a tactic set* Γ*, if* $\Pi \models \Psi$ *and* $\forall(R,pc) \in \Gamma.\ \forall(\pi',pc') \in \Pi.\ pc = pc' \implies \Phi(\pi') \in L(R)$ *then* $\forall(R,pc) \in \Gamma.(Fwd,Reach) \models \Psi_T(R,pc)$.

Proof. **Assume** $\exists(R,pc) \in \Gamma$ such that $(\pi,pc) \in \Pi$ and
$(Fwd,Reach) \not\models \Psi_T(R,pc)$. Consider the three types of tactics for R:
Type 1: $R = \neg(l_{src}.^i.^* l_{dst})$
$(Fwd,Reach) \not\models \Psi_T(R,pc) \implies (Fwd,Reach) \not\models \forall sw,k \geq i+1.(sw,pc,k) \notin Reach$
Therefore $\exists sw,k \geq i+1.(sw,pc,k) \in Reach$.
Therefore, $\Phi(\pi) \in L(l_{src}\ .^i\ .^*\ \phi(sw)\ .^*\ l_{dst}) \implies \Phi(\pi) \in L(l_{src}.^i.^* l_{dst})$.
Contradiction, as $\Phi(\pi) \in L(R)$.

Type 2: $R = \neg(l_{src}.^i\ l\ .^* l_{dst})$
$(Fwd,Reach) \not\models \Psi_T(R,pc) \implies (Fwd,Reach) \not\models$
$\forall sw.\ \phi(sw) = l \wedge sw \neq dst \implies (sw,pc,i+1) \notin Reach$.
Therefore $\exists sw.\ sw \neq dst \wedge \phi(sw) = l \wedge (sw,pc,i+1) \in Reach$.
Therefore, $\Phi(\pi) \in L(l_{src}\ .^i\ l\ .^*\ l_{dst})$.
Contradiction, as $\Phi(\pi) \in L(R)$.

Type 3: $R = \neg(l_{src}.^i\ l_1\ l_2\ .^* l_{dst})$
$(Fwd,Reach) \not\models \Psi_T(R,pc) \implies (Fwd,Reach) \not\models$
$\forall n_1,n_2.\ \phi(n_1) = l_1 \wedge \phi(n_2) = l_2 \wedge n_2 \neq dst \implies \neg(Reach(n_1,pc,i+1) \wedge Fwd(n_1,n_2,pc))$.
Therefore $\exists n_1,n_2.\ \phi(n_1) = l_1 \wedge \phi(n_2) = l_2 \wedge n_2 \neq dst \wedge Reach(n_1,pc,i+1) \wedge Fwd(n_1,n_2,pc)$.
Therefore, $\Phi(\pi) \in L(l_{src}\ .^i\ l_1\ l_2\ .^*\ l_{dst})$.
Contradiction, as $\Phi(\pi) \in L(R)$. □

Appendix C

Zeppelin Proofs

C.1 Intra-domain Configuration Synthesis Proofs

Theorem C.1.1 (Hardness of synthesis)**.** *Given a network with a single domain, and a positive number k, the problem generating a path-compliant configuration with at most k static routes is NP-complete.*

Proof. We show that the decision version of the minimum vertex cover problem, i.e., there exists a vertex cover of size $\leq k$, which is NP-complete, reduces to finding a set of static routes of size $\leq k$ and OSPF weights for a network with only one domain. The latter is also in NP, so after the reduction we can conclude that it is also NP-complete.

Let $G = (V, E)$ be an instance of the minimum vertex cover problem. A set of vertices $VC \subseteq V$ is the vertex cover if $\forall (v_1, v_2) \in E.\ v_1 \in VC \vee v_2 \in VC$.

We now show how to construct a topology $T = (R, L)$ and a corresponding set of paths Π that can be enforced by configuration C which requires static routes SR such that $|SR| \leq k$ if and only if the corresponding $VC(SR)$ is a vertex cover of the graph G and $|VC(SR)| \leq k$.
Construction. For every vertex $v \in V$: add a vertex r_v. For every edge $(u, v) \in E$: add two vertices s_{uv} and t_{uv} to R. Add edges connecting $s_{uv} \to r_u$, $s_{uv} \to r_v$, $r_u \to t_{uv}$ and $r_v \to t_{uv}$. Figure C.1 illustrates this construction.

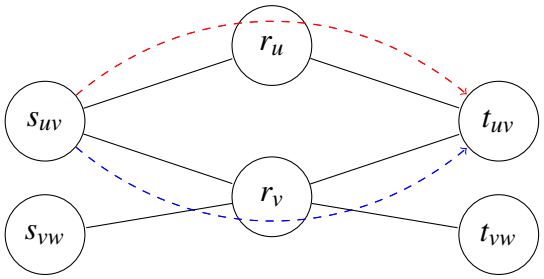


Fig. C.1 Construction for reduction to Vertex Cover.

If there is another edge $(v,w) \in E$, then s_{vw} and t_{vw} have an edge connecting to r_v (shown in Figure C.1).

For each edge $(u,v) \in E$, we add two paths in Π: $s_{uv} \rightarrow r_u \rightarrow t_{uv}$ for destination host d_u and $s_{uv} \rightarrow r_v \rightarrow t_{uv}$ for destination host d_v. (dashed paths in Figure C.1).

We now prove that if there exists a set of static routes SR such that $|SR| \leq k$ such that the resulting configurations are path-compliant for Π, then there exists a vertex cover VC of G such that $|VC| \leq k$.

For each static route $sr \in SR$, the static route has to be placed at the source, either going to r_u or r_v (structure of topology T). We construct a set $VC(SR)$ by adding the vertex v based on the endpoints of each static route $sr \in SR$. To show that $VC(SR)$ is a vertex cover of G, we first prove Lemma 3.

Lemma 3. *For each diamond formed by the input paths, atleast 1 static route on one of the edges of the paths of the diamond is required to find a valid solution to the OSPF edge weights.*

Proof. Given two paths π_1 and π_2 for destinations d_u and d_v, we define these paths form a diamond if these paths intersect at two routers (s_{uv} and t_{uv}) without any common router in between. Consider the following diamond constructed by paths π_1: $s_{uv} \rightarrow r_u \rightarrow t_{uv}$ for destination d_u and π_2: $s_{uv} \rightarrow r_v \rightarrow t_{uv}$ for destination d_v. Let us assume there exists a solution for the OSPF edge weights without any static routes.

We add the following inequality to make π_1 is the shortest path from s_{uv} to t_{uv} by ensuring π_1 is shorter than the path from s_{uv} to t_{uv} via r_v:

$$W(s_{uv},r_u) + W(r_u,t_{uv}) < W(s_{uv},r_v) + W(r_v,t_{uv}) \tag{C.1}$$

Since π_2 is also the shortest path from s_{uv} to t_{uv}, the linear inequality added is:

$$W(s_{uv},r_v) + W(r_v,t) < W(s_{uv},r_u) + W(r_u,t_{uv}) \tag{C.2}$$

Since there are no static routes on the edges of π_1 and π_2, none of the above equations are eliminated. Adding equations C.1 and C.2 yields the inequality $0 < 0$, which is inconsistent and therefore, no solution to the edge weights exists for this system of equations, which contradicts our assumption. Therefore, for each diamond formed by the input paths, atleast 1 static route on one of the edges of the paths of the diamond is required to find a valid solution to the OSPF edge weights. □

For every edge $(u,v) \in E$, the constructed paths from s_{uv} to t_{uv} form a diamond. Thus, by Lemma 3, the diamond created by the paths corresponding to each edge in G requires atleast one static route to eliminate the inconsistency caused by the diamond. If a static route's endpoints contains r_u, we put u in $VC(SR)$ and similarily for r_v. Edge (u,v) is covered since atleast one static route is added, thus, atleast one of $\{u,v\}$ is in $VC(SR)$. Thus, if SR eliminates all diamond inconsistencies to find a solution to the OSPF weights, the corresponding set $VC(SR)$ covers all edges in E. Therefore, $VC(SR)$ is a vertex cover.

Thus, by finding a set of static routes SR such that $|SR| \leq k$ such that all the diamond inconsistencies are eliminated, and there exists OSPF weights W such that the configurations forward traffic along Π, we can find a vertex cover VC for graph G such that $|VC| \leq k$.

This transformation is polynomial, the constructed network topology T has $|V|+2|E|$ nodes, $4|E|$ links and $2|E|$ paths. Therefore, OSPF configuration synthesis with number of static routes $\leq k$ is NP-complete. Thus, OSPF synthesis with minimal number of static routes is NP-hard. □

Theorem C.1.2 (OSPF Synthesis Soundness)**.** *For a set of paths Π, if edge weights W satisfy constraints (4.1) and (4.2), then each path $\pi \in \Pi$ is the unique shortest path between its endpoints.*

Proof. Let us assume there exists a path $\pi = (s,s_1)(s_1,s_2)\dots(s_n,d)$ in Π such that π is *not* the unique shortest path between s and d. Let $\sigma = (s,r_1)(r_1,r_2)\dots(r_m,d)$ be one of the shortest path that is different from π. Since, σ is the shortest weighted path:

$$\sum_{\pi} W \geq \sum_{\sigma} W \tag{C.3}$$

Let $s_i \neq r_i$ be the first point of divergence of the paths—i.e., for every $j < i$, $s_j = r_j$. Constraints (4.2) impose $(s_{i-1},s_i)\dots(s_n,d)$ to be the unique shortest path by ensuring the edge weights of the path are smaller than any path going through a neighbouring router other than s_i. Consider the neighbour router r_i in σ:

$$\sum_{(s_{i-1},s_i)\cdots(s_n,d)} W < W(s_{i-1},r_i) + D(r_i,d)$$

We can then use constraint (4.1) to expand the term $D(r_k,d)$ for $i \leq k \leq m$ and obtain the following:

$$\sum_{(s_{i-1},s_i)\cdots(s_n,d)} W < W(s_{i-1},r_i) + W(r_i,r_{i+1}) + D(r_{i+1},d)$$

$$\cdots$$

$$\sum_{(s_{i-1},s_i)\cdots(s_n,d)} W < W(s_{i-1},r_i) + W(r_i,r_{i+1}) + \ldots W(r_m,d) + D(d,d)$$

$$\sum_{(s_{i-1},s_i)\cdots(s_n,d)} W < \sum_{(r_{i-1},r_i)\cdots(r_m,d)} W$$

Adding $\Sigma_{(s,s_1)\cdots(s_{i-2},s_{i-1})} W$ to both sides:

$$\sum_{\pi} W < \sum_{\sigma} W$$

This contradicts assumption (C.3) that σ is one of the shortest paths from s to d. Hence, proved, all paths in Π are the unique shortest paths if W satisfy constraints (4.1) and (4.2). □

Theorem C.1.3 (OSPF Synthesis Completeness)**.** *For a set of paths* Π *and edge weights* W*, if every path* $\pi \in \Pi$ *is the unique shortest path between its endpoints, then there exist values for* D *for which* W *satisfies constraints (4.1) and (4.2).*

Proof. Assume we are given W and Π. We show that if we assign to each $D(s,t)$ the weight of the shortest path from s to t according to W, both constraints (4.1) and (4.2) hold.

The distance constraints (4.1) are satisfied using the triangular inequality: If r lies on the shortest path from s to t, $D(s,t) = W(s,r) + D(r,t)$, otherwise s, r and t form a triangle and $D(s,t) \leq W(s,r) + D(r,t)$.

Since, every path $\pi \in \Pi$ is the shortest unique path, each subpath $\sigma = (s,r_1)\ldots(r_n,d)$ of π to the destination is also the unique shortest path. By definition of unique shortest path, $\sum_\sigma W$ is smaller than the shortest path from s to d through r' not in σ—i.e., $\sum_\sigma W < W(s,r') + D(r',d)$. Hence, W satisfies constraints (4.2). □

Theorem C.1.4 (OSPF+SR Soundness)**.** *Given a set of packet classes* PC *and paths* Π*, Algorithm 4 outputs a configuration* $C(W,SR)$ *such that* $\Pi \subseteq \mathfrak{P}^C(PC)$.

Proof. Let us assume there exists a packet class pc in PC whose path $\pi_{pc} = (s_{pc},s_1)\ (s_1,s_2)\ \ldots(s_n,d_{pc})$ is not induced by C—i.e., $\pi_{pc} \notin \mathfrak{P}^C(pc)$. Let the destination IP of pc be λ_{pc}.

Given the routing function $\mathfrak{R}^C$ constructed from SR and W (§4.2.1), let the first router where routing diverges from π_{pc} be s_p—i.e., $s_{p+1} \notin \mathfrak{R}^C(s_p,\lambda_{pc})$. Let $r \in \mathfrak{R}^C(s_p,\lambda_{pc})$. Algorithm 4 on line 3 adds the following constraint to ensure the sub-path of π_{pc} from s_p to d_{pc} is

the unique shortest path:

$$\sum_{(s_p,s_{p+1})\ldots(s_n,d_{pc})} W(s_i,s_j) < W(s_p,r) + D(r,d_{pc}) \tag{C.4}$$

Algorithm 4 only removes a subset of the shortest path constraints and not the distance constraints (9). We consider two cases: when Algorithm 4 does not remove Constraint (C.4) and when it does.

Case 1:. Algorithm 4 does not remove Constraint (C.4). Thus, there is no static route (s_p, s_{p+1}) for λ_{pc} (line 8), and OSPF-based forwarding occurs at s_p. For any path $(r,r_1),(r_1,r_2)$ $\ldots(r_m,d_{pc})$, using the distance constraints (4.1) to expand $D(r,d_{pc})$, we get the following:

$$\sum_{(s_p,s_{p+1})\ldots(s_n,d_{pc})} W(s_i,s_j) < W(s_p,r) + W(r,r_1) + D(r_1,d_{pc})$$

$$\ldots$$

$$\sum_{(s_p,s_{p+1})\ldots(s_n,d_{pc})} W(s_i,s_j) < W(s_p,r) + W(r,r_1) + \ldots W(r_m,d_{pc}) + D(d_{pc},d_{pc})$$

Therefore, the weight of any path through r is greater than the path through s_p and, since OSPF sends traffic through the shortest weighted path, r cannot be in $R^C_{ospf}(s_p,\lambda_{pc})$. Hence a contradiction.

Case 2:. Algorithm 4 removes Constraint (C.4) and $SR(s_p,\lambda_{pc}) = \{s_{p+1}\}$ (lines 8-9). Thus, $s_{p+1} \in \mathfrak{R}^C(s_p,\lambda_{pc})$ as static routes have the higher priority than OSPF. This contradicts our assumption that $s_{p+1} \notin \mathfrak{R}^C(s_p,\lambda_{pc})$. □

C.2 Policy-Resilient Configuration Synthesis Proofs

Theorem C.2.1 (OSPF Waypoint Soundness)**.** *Given a set of waypoint paths* $\{(\pi_{pc},\mathbb{W}_{pc}) \mid pc\}$*, if edge weights W satisfy constraints (4.3) and (4.4), for each packet class pc, the shortest path between its endpoints traverses one of the waypoints in* $\mathbb{W}_{pc}$*.*

Proof. Let us assume there exists a packet class pc with waypoint set $\mathbb{W}_{pc}$ and waypoint path $\pi_{pc} = (s_{pc},s_1)(s_1,s_2)\ldots(s_n,d_{pc})$, such that the shortest path $\sigma_{pc} = (s_{pc},r_1)\,(r_1,r_2)\ldots(r_m,d_{pc})$ is not waypoint-compliant—i.e., for every i, we have $r_i \notin \mathbb{W}$. Since σ_{pc} is the shortest weighted path:

$$\sum_{\pi_{pc}} W \geq \sum_{\sigma_{pc}} W \tag{C.5}$$

Let $s_i \neq r_i$ be the first point of divergence of the paths—i.e., for every $j < i$, $s_j = r_j$. Constraints (4.4) impose $(s_{i-1},s_i)\ldots(s_n,d)$ to be shorter than any path which is not waypoint-

compliant. Let us consider the neighbour router r_1 in σ_{pc}:

$$\sum_{(s_{i-1},s_i)\cdots(s_n,d)} W < W(s_{i-1},r_i) + D(r_i,d,\mathbb{W}_{pc})$$

Since σ_{pc} does not traverse any waypoint in $\mathbb{W}_{pc}$, we use constraints (4.3) to expand the terms $D(r_k,d_{pc},\mathbb{W}_{pc})$ for $i \leq k \leq m$:

$$\sum_{(s_{i-1},s_i)\cdots(s_n,d)} W < W(s_{i-1},r_i) + W(r_i,r_{i+1}) + D(r_{i+1},d,\mathbb{W}_{pc})$$

$$\cdots$$

$$\sum_{(s_{i-1},s_i)\cdots(s_n,d)} W < W(s_{i-1},r_i) + W(r_i,r_{i+1}) + \ldots W(r_m,d) + D(d,d,\mathbb{W}_{pc})$$

$$\sum_{(s_{i-1},s_i)\cdots(s_n,d)} W < \sum_{(r_{i-1},r_i)\cdots(r_m,d)} W$$

Adding $\Sigma_{(s,s_1)\cdots(s_{i-2},s_{i-1})}$ to both sides:

$$\sum_{\pi_{pc}} W < \sum_{\sigma_{pc}} W$$

However, this contradicts the assumption (C.5) that σ_{pc} is the shortest path from s_{pc} to d_{pc}. □

Theorem C.2.2 (OSPF+SR Waypoint Soundness)**.** *Given a set of waypoint paths* $\{(\pi_{pc},\mathbb{W}_{pc}) \mid pc\}$*, Algorithm 5 outputs a configuration* $C(W,SR)$ *such that, for every packet class pc, there exists a path in* $\mathfrak{P}^C(pc)$ *that traverses one of the waypoints in* $\mathbb{W}_{pc}$*.*

Proof. Let us assume there exists a packet class pc with destination λ, waypoint set $\mathbb{W}_{pc}$, and waypoint path $\pi_{pc} = (s_{pc},s_1)(s_1,s_2)\ldots(s_n,d_{pc})$, such that for pc, there exists no path in $\mathfrak{P}^C(pc)$ which is waypoint-compliant. There are two cases: either there exists no path in $\mathfrak{P}^C(pc)$ or the paths do not traverse any waypoint in $\mathbb{W}_{pc}$.

Case 1:. $\mathfrak{P}^C(pc) = \emptyset$—i.e., there is a routing loop caused by static routes (OSPF forwarding is loop-free). We denote the set of static routes for λ by $SR(\lambda)$. Note that multiple packet classes can share the same destination IP and destination router, and we construct a destination-based tree from these paths, denoted by ξ_λ

Let us denote the routing loop as $(r_0,r_1)(r_1,r_2)\ldots(r_{n-1},r_0)$. Since, the path is a loop with atleast one static route, there must exist one router r_i in the loop such that $(r_{i-1},r_i) \in SR(\lambda)$ and the next router in the loop r_j which has a static route $(r_j,r_{j+1}) \in SR(\lambda)$ and r_j is not downstream to r_i in ξ_λ (meaning there is no directed path from r_i to r_j in ξ_λ).

Consider r_i. The path from r_i to r_j is due to OSPF forwarding, therefore, there exists a path $\sigma = (r_i, r_{i+1}) \dots (r_{j-1}, r_j)(r_j, l_1) \dots (l_o, d_{pc})$ which is the shortest path from r_i to d_{pc}.

Since static routes are only added on edges in ξ_λ, let us denote the path from r_i to d_{pc} in ξ_λ as $(r_i, u_1)(u_1, u_2) \dots (u_p, d_{pc})$. Consider constraint (4.5) which is added for static route (r_{i-1}, r_i) and non-downstream router r_j:

$$\sum_{(r_i,u_1)\dots(u_p,d_{pc})} W < D(r_i, r_j) + D(r_j, d_{pc})$$

Using distance constraints (4.1), we can expand $D(r_i, r_j)$ and $D(r_j, d_{pc})$ along the paths $(r_i, r_{i+1}) \dots (r_{j-1}, r_j)$ and $(r_j, l_1) \dots (l_o, d_{pc})$ respectively.

$$\sum_{(r_i,u_1)\dots(u_p,d_{pc})} W < \sum_{(r_i,r_{i+1})\dots(r_{j-1},r_j)} W + \sum_{(r_j,l_1)\dots(l_o,d_{pc})} W$$

However, this contradicts the assumption that σ is the shortest path from r_i to d_{pc}.

Case 2:. All paths in $\mathfrak{P}^C(pc) \neq \emptyset$ are not waypoint-compliant. Let $\sigma_{pc} = (s_{pc}, r_1)(r_1, r_2) \dots (r_n, d_{pc})$ be one such path in $\mathfrak{P}^C(pc)$. Given the routing function $\mathfrak{R}^C$ constructed from *SR* and *W* (§4.2.1), let the first router where routing diverges from π_{pc} be s_p—i.e., $s_{p+1} \notin \mathfrak{R}^C(s_p, \lambda_{pc})$. Algorithm 5 on line 3 adds the following constraint to ensure the sub-path of π_{pc} from s_p to d_{pc} is shorter than non-waypoint paths:

$$\sum_{(s_p,s_{p+1})\cdots(s_n,d_{pc})} W < W(s_p, r_{p+1}) + D(r_{p+1}, d_{pc}, \mathbb{W}_{pc}) \tag{C.6}$$

Algorithm 5 only removes a subset of the waypoint and loop constraints constraints and not the distance constraints (10). We further consider two sub-cases: whether Algorithm 5 removes Constraint (C.6) or not.

Case 2A: Algorithm 5 does not remove Constraint (C.6). Thus, there is no static route (s_p, s_{p+1}) for λ_{pc} (line 9), and OSPF-based forwarding occurs at s_p. Since σ_{pc} does not traverse any waypoint in $\mathbb{W}_{pc}$, we use constraints (4.3) to expand the terms $D(r_k, d_{pc}, \mathbb{W}_{pc})$ for $p+1 \leq k \leq m$:

$$\sum_{(s_p,s_{p+1})\cdots(s_n,d_{pc})} W < W(s_p, r_{p+1}) + W(r_{p+1}, r_{p+2}) + D(r_{p+2}, d_{pc}, \mathbb{W}_{pc})$$

$$\dots$$

$$\sum_{(s_p,s_{p+1})\cdots(s_n,d_{pc})} W < W(s_p, r_{p+1}) + \dots + W(r_m, d_{pc}) + D(d_{pc}, d_{pc}, \mathbb{W}_{pc})$$

By adding weights of $(s_{pc}, s_1) \ldots (s_{p-1}, s_p)$, we obtain that weight of σ_{pc} is greater than π_{pc}.

Case 2A.1: There are no static routes on σ_{pc}. This is a contradiction as OSPF has a shorter path π_{pc} to send traffic to, instead of σ_{pc}.

Case 2A.2: σ_{pc} has a static route on the path. Since, Algorithm 5 only adds static routes on links in ξ_λ, σ_{pc} intersects ξ_λ again after diverging at r_p. Let the next divergence from ξ_λ be at r_q. Using Case 2 argument at r_q, we can reach a contradiction. This is because, the number of routers at which σ_{pc} intersect with ξ_λ would be bounded, and at the last divergence, there would be no static routes to the destination.

Case 2B: Algorithm 5 removes Constraint (4.4) and $SR(s_p, \lambda_{pc}) = \{s_{p+1}\}$ (lines 9-10). Thus, $s_{p+1} \in \mathfrak{R}^C(s_p, \lambda_{pc})$ as static routes have the higher priority than OSPF. This contradicts our assumption that $s_{p+1} \notin \mathfrak{R}^C(s_p, \lambda_{pc})$.

Hence, for every packet class, there exists a path in the set of induced paths which traverses through one of the waypoints. □

Theorem C.2.3 (OSPF 2-Waypoint Soundness)**.** *For a set of edge-disjoint waypoint paths* $\{(\pi^1_{pc}, \pi^2_{pc}, \mathbb{W}_{pc}) \mid pc\}$*, if edge weights W satisfy constraints (4.3), (4.4) and (4.6), then for any arbitrary single link failure, the shortest path between each packet class's endpoints traverses one of the waypoints in* $\mathbb{W}_{pc}$*.*

Proof. Let us assume there exists a packet class pc with waypoint set $\mathbb{W}_{pc}$ and edge-disjoint waypoint paths $\pi^1_{pc} = (s_{pc}, s_1)(s_1, s_2) \ldots (s_n, d_{pc})$, and $\pi^2_{pc} = (s_{pc}, t_1)(t_1, t_2) \ldots (t_l, d_{pc})$; and there exists a link l failure such that for pc, the shortest path $\sigma_{pc} = (s_{pc}, r_1)\ (r_1, r_2) \ldots (r_m, d_{pc})$ is not waypoint-compliant——i.e., for every i, we have $r_i \notin \mathbb{W}$.

A single link l failure cannot disable both π^1_{pc} and π^2_{pc} as they are edge-disjoint. **Case 1:.** Link l is not in path π^1_{pc}. Since, σ_{pc} is the shortest path:

$$\sum_{\sigma_{pc}} W \leq \sum_{\pi^1_{pc}} W$$

Constraint (4.6) imposes the following:

$$\sum_{\pi^1_{pc}} W < D(s_{pc}, d_{pc}, \mathbb{W}_{pc})$$

Since, σ_{pc} does not traverse any waypoint, we can use constraints (4.3) for expanding $D(r_k, d_{pc}, \mathbb{W}_{pc})$ for $1 \leq k \leq m$:

$$\sum_{\pi^1_{pc}} W < W(s_{pc}, r_1) + D(r_1, d_{pc}, \mathbb{W}_{pc})$$

$$\ldots$$

$$\sum_{\pi^1_{pc}} W < W(s_{pc}, r_1) + W(r_1, r_2) + \ldots + W(r_m, d_{pc}) + D(d_{pc}, d_{pc}, \mathbb{W}_{pc}) = \sum_{\sigma_{pc}} W$$

This contradiction the assumption that σ_{pc} is the shortest path from s_{pc} to d_{pc}.
Case 2:. Link l is not in path π^2_{pc}. Proof same as Case 1 for π^2_{pc}. **Case 3:.** Link l is not in path π^1_{pc} or π^2_{pc}. Using Theorem C.2.1, σ_{pc} cannot be the shortest path as it does not traverse any waypoint in $\mathbb{W}_{pc}$.

Hence proved, under any arbitrary single-link failure, the shortest path between each packet class's endpoints traverses one of the waypoints in $\mathbb{W}_{pc}$. □

C.3 Domain Assignment Proofs

Theorem C.3.1 (Hardness of domain assignments)**.** *Given a network and set of paths* Π*, the problem of generating a domain assignment for which there exists a configuration that is path-compliant with* Π *with no static routes is NP-complete.*

Proof. We show the k-graph coloring problem, which is NP-complete reduces to finding a domain assignment such that the number of static routes is 0. The latter is also in NP, so after the reduction we can conclude that it is also NP-complete.

Let $G = (V, E)$ be an instance of the k-graph coloring problem. Formally, we need to find a coloring $C : V \mapsto \{1, 2, \ldots k\}$ such that for $\forall (u, v) \in E . C(u) \neq C(v)$.

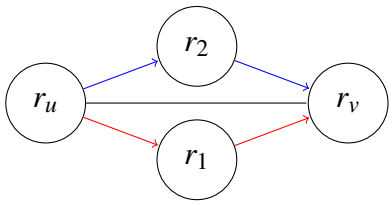


Fig. C.2 Construction for reduction to Graph Coloring.

Construction. Let us consider a network topology $T = (R, L)$. For each $v \in V$, add a router $r_v \in R$. To ensure any domain assignment to the r_v's is valid, we add link to connect every router (each domain must be contiguous).

For every edge $(u, v) \in E$, we add r_1, r_2 to R, and add paths: $r_u \to r_1 \to r_v$ and $r_u \to r_2 \to r_v$ with different destinations to Π. We can notice that these two paths form a diamond.

Suppose we find a domain assignment Θ with k domains such that the number of static routes used is zero. Since, the count of static routes is 0, for two routers r_u, r_v which have a diamond, $\Theta(r_u) \neq \Theta(r_v)$. This is because, if $\Theta(r_u) = \Theta(r_v)$, then atleast one static route would be required to eliminate the diamond (Lemma 3). Therefore, for every edge in $(u, v) \in E$, the routers r_u and r_v belong to different domains, therefore u and v have different colors. There, the k-graph coloring problem reduces to finding a domain assignment with zero static routes. Therefore, the path-compliance synthesis problem is NP-complete. □