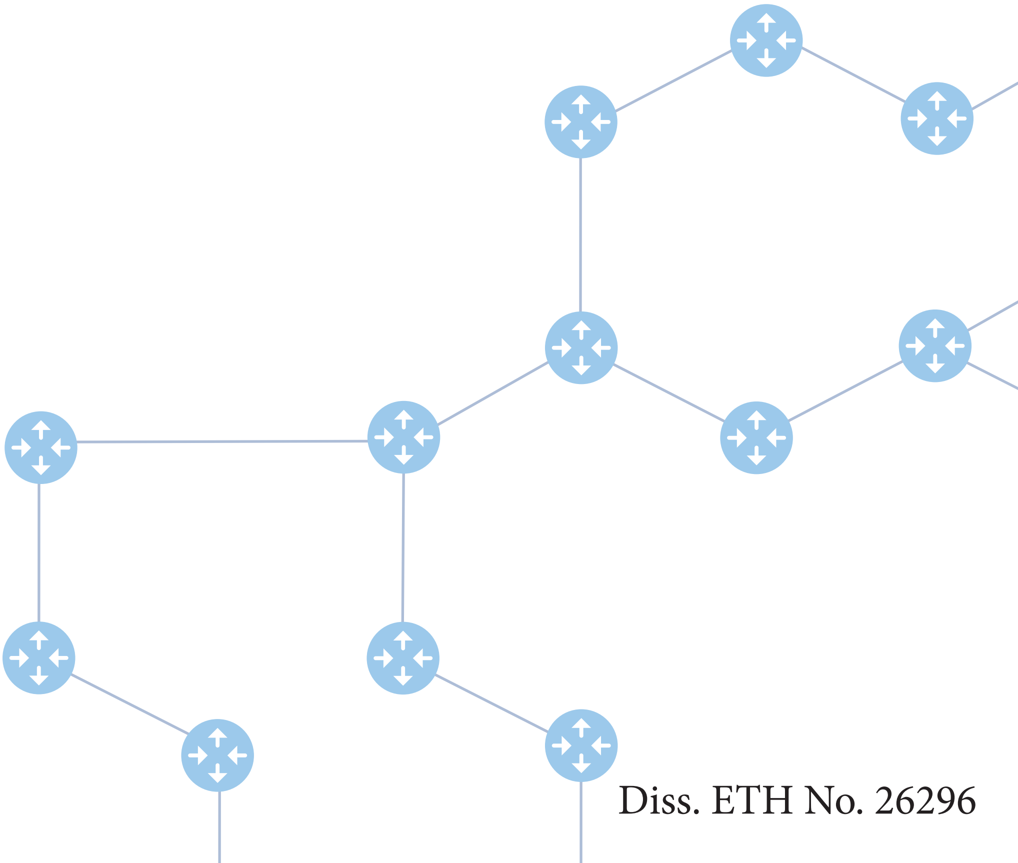
Ahmed Elhassany

Towards reliable network control planes



Diss. ETH No. 26296

DISS. ETH NO. 26296

TOWARDS RELIABLE NETWORK CONTROL PLANES

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZÜRICH
(Dr. sc. ETH Zürich)

presented by

AHMED H.A. ELHASSANY
Master of Science
University of Delaware

born on 1st April 1986
citizen of Palestine

accepted on the recommendation of

Prof. Dr. Laurent Vanbever
Prof. Dr. Aditya Akella
Prof. Dr. Arvind Krishnamurthy
Prof. Dr. Ankit Singla
Prof. Dr. Martin Vechev

2019

Ahmed H.A. Elhassany: *Towards Reliable Network Control Planes,* © 2019

DOI: 10.3929/ethz-a-26296

To Mom and Dad

ABSTRACT

Many critical services, such as e-commerce, emergency response, or even remote surgeries, rely on computer networks. This strategic importance makes them a mission-critical infrastructure that must operate reliably, with minimum downtime. Achieving reliable operations is challenging, though, as confirmed by the countless major network downtimes that are frequently making the news.

In this dissertation, we develop techniques and tools to enable the reliable network operations of the two main types of networks deployed today: *(i)* networks running distributed routing protocols; and *(ii)* networks managed by a logically-centralized controller, also known as Software-Defined Networks (SDN). Each paradigm comes with its own set of operational challenges. In networks running distributed routing protocols, the main problem is configuring each device correctly. Studies have indeed shown that human-induced misconfigurations, *not* physical failures, explain the majority of network downtimes. In contrast, in SDN, the main problem is ensuring the correctness of the controller software. Therefore, we tackle two complementary problems: *(i) how to synthesize configurations for networks running distributed routing protocols automatically?*; and *(ii) how to catch bugs in SDN networks?*

To eliminate human-induced misconfigurations, we propose two novel and complementary techniques to synthesize configurations from high-level policies. The first technique aims at generality, while the second one aims at usability and scalability. Specifically, the first technique enables the synthesis of correct configurations for arbitrary protocols from operators' intents and a formal specification of the protocols' behavior. While useful, we show that this technique is limited in its scalability and can produce configurations that are far from what a human would write. The second set of techniques rely on autocompletion to address these problems. Here, we allow the network operators to specify parts of the configurations and constraint the values that the synthesizer is permitted to produce. We show that autocompletion enables the synthesizer to scale to large networks and generates interpretable configurations.

To catch bugs in SDN controllers, we develop new techniques for detecting and troubleshooting concurrency violations. As an event-based system, an SDN controller is indeed particularly subject to concurrency issues. Thus, we present a sound and complete dynamic concurrency analyzer for SDN controllers. Furthermore, we present techniques to assist the developers in identifying and troubleshooting the source of the reported concurrency violations.

ZUSAMMENFASSUNG

Viele Dienstleistungen aus unserem Alltag, wie z.B. der elektronische Zahlungsverkehr, Alarmierungssysteme oder chirurgische Eingriffe mit ferngesteuerten Robotern benötigen heutzutage Computer Netzwerke, um richtig zu funktionieren. Es ist deshalb umso wichtiger, dass die verwendeten Netzwerke möglichst ohne Fehler und Unterbrechungen arbeiten. Unzählige Berichte zu schwerwiegenden Netzwerk Ausfällen zeigen jedoch, wie schwierig es ist, Netzwerke zuverlässig zu betreiben.

In dieser Dissertation erarbeiten wir neue Verfahren und Hilfsmittel, um Computer Netzwerke verlässlich und ohne Ausfälle zu betreiben. Dabei kann heutzutage zwischen zwei Netzwerk-Architekturen unterschieden werden: Einerseits traditionelle Netzwerke, die auf verteilten Routingprotokollen basieren; und andererseits sogenannte Software-Defined Netzwerke (SDN), welche von einem zentralen Controller gesteuert werden. Die beiden Netzwerk-Architekturen haben je ihr eigenen Probleme und Schwierigkeiten. In traditionellen Netzwerken besteht die grösste Schwierigkeit in der korrekten Konfiguration aller Geräte im Netzwerk, da jedes Gerät einzeln konfiguriert werden muss. Wie Studien zeigen, ist der Grund für die meisten Netzwerk Unterbrüche nicht etwa ein Hardwarefehler, sondern menschliches Versagen bei der Gerätekonfiguration. SDN Netzwerke bringen ganz andere Herausforderungen mit sich. In diesen Netzwerken müssen wir vor allem sicherstellen, dass der zentrale Controller fehlerfrei läuft. Zusammenfassend gehen wir deshalb zwei Probleme an, die sich gegenseitig ergänzen: Wie können wir automatisch Gerätekonfigurationen für traditionelle Netzwerke generieren?; und wie können wir Softwarefehler in SDN Netzwerken finden?

Um menschliche Konfigurationsfehler zu vermeiden, präsentieren wir zwei neue, sich ergänzende Verfahren, um die Konfiguration aufgrund von abstrakten Richtlinien automatisch herzustellen. Das erste Verfahren generiert Konfigurationen, die möglichst allgemein einsetzbar sind, wobei der zweite Ansatz mehr darauf achtet, dass die erzeugten Konfiguration für den Netzwerkbetreiber verständlich und skalierbar sind. Basierend auf den Zielen des Netzwerkbetreibers und auf einer formellen Protokoll-Beschreibung, kann das erste Verfahren korrekte Konfigurationen für beliebige Protokolle herstellen. Die so erzeugten Konfiguration sind jedoch oft schwer verständlich und der Ansatz kommt an seine Grenzen mit zunehmender Grösse des Netzwerkes. Anders sieht es beim zweiten Ansatz aus. Ein Netzwerkbetreiber kann nun bereits wichtige Teile der Konfiguration vorgeben oder die Werte verschiedener Parameter einschränken. Unser Verfahren vervollständigt dann automatisch die fehlenden Teile der Konfiguration. Wie wir in verschiedenen Auswertungen zeigen, kann dieser Ansatz Konfigurationen für sehr grosse Netzwerke generieren, die zudem einfach zu verstehen sind.

Um Softwarefehler in SDN Controller zu finden, erarbeiten wir neue Ansätze, um Probleme in nebenläufigen Prozessen zu erkennen und zu lösen. Solche Probleme treten in einem SDN Controller häufig auf, da es sich um ein ereignisgesteuertes System handelt. Deshalb präsentieren wir ein System, das SDN Controller dynamisch analysieren kann und beweisen dessen Korrektheit und Vollständigkeit. Zudem beschreiben wir Verfahren, die es den Entwicklern ermöglichen, die zugrundeliegende Ursache des Nebenläufigkeitsproblems zu finden und zu lösen.

PREAMBLE

Many critical services such as teleconferencing, e-commerce, digital currencies, emergency response, and remote surgeries rely on computer networks. This strategic importance makes them a mission-critical infrastructure that must operate reliably. Achieving reliable operations is a challenging task, as confirmed by the countless major network downtimes that are frequently making the news [1–9].

To ensure reliable operations, network operators need to guarantee that the network is operating correctly; i.e., it is forwarding data according to its business and performance goals. The forwarding paths along which the network devices forward traffic (i.e., the application data) are computed by the network's *control-plane*. Nowadays, two different paradigms implement a network's control-plane: *distributed* using distributed routing protocols and *logically centralized* using Software-Defined Networking (SDN). Therefore, we address both paradigms' open problems to ensure that networks implement the operators' intents correctly. Namely, we develop new methods and techniques to *(i) correctly configure distributed routing protocols using synthesis*, and *(ii) verify that SDN networks are free from concurrency errors*.

The first problem we tackle in this dissertation is how to produce correct low-level configurations automatically such that networks running distributed routing protocols behave according to the operators' intents. In such networks, a network device (i.e., *router*) runs software implementing the routing protocols. This software operates by exchanging reachability information with the device's neighbors and then using this reachability information and local configurations (i.e., low-level instructions that are the inputs to the routing protocols) to decide how to forward traffic. The network operators derive (often manually) the low-level configurations from high-level business and performance requirements. They have to derive the local configurations for hundreds or even thousands of network devices. A single *misconfiguration* (i.e., a configuration under which the network does not meet the expected behavior) not only can violate the business and performance requirements but also can bring down the network infrastructure, or worse a piece of the Internet. As an illustration, Facebook suffered from widespread issues for about an hour due to a misconfiguration [1], while a misconfiguration in Verizon's network slowed down the performance of the Internet for two hours [2]. Studies show that human-induced misconfigurations, *not* physical failures, explain the majority of network downtimes [3, 10, 11].

In this dissertation, we propose *network-wide configurations synthesis* as an alternative to manually configuring networks running distributed routing protocols. Synthesizing configurations not only simplifies the task of configuring a network but also reduces the chances of human-induced errors and increases the network's overall reliability. Designing a network-wide configurations synthesis framework that is *practical*, *expressive*, and *scalable* to production-size networks is a challenging task. The synthesis framework needs to capture complex distributed routing protocols computations and invert them to find inputs (configuration parameters) that induce specific forwarding paths. Then, the synthesis framework needs to efficiently search the large space of possible configuration parameters and find configurations (for every single device in the network) that meet the operators' intents.

In this dissertation, we develop two novel techniques to synthesize network-wide configurations. The first technique is a general synthesis framework that is capable of synthesizing configurations for any distributed routing protocol based on formal specifications of the protocol's behavior. In particular, our general synthesizer takes two inputs: *(i)* the operators' intents; and *(ii)* the formal specifications of the network (i.e., how the distributed routing protocols computes the paths to forward traffic). However, the generality comes at the cost of interpretability and scalability. Specifically, a general synthesizer cannot produce configurations that follow the best practices for a given protocol to improve the interpretability of the synthesized configurations, and cannot leverage protocol-specific optimizations to improve the scalability of the synthesizer. We address the interpretability and scalability limitations in the second technique we develop in this dissertation. The second technique frames the network-wide configurations synthesis problem as an autocompletion problem. In particular, our autocompletion synthesizer takes two inputs: *(i)* the operators' intents; and *(ii)* a sketch of the desired output configurations. The configurations sketch enables the network operators to specify the parts of the existing configurations that the synthesizer is permitted to change and to constrain the values that the synthesizer is permitted to use. This approach enables the synthesizer to scale to large networks and produce configurations that are similar to what a human operator would write.

The second problem we tackle in this dissertation is *detecting and troubleshooting concurrency bugs in SDN controllers*. SDN argues for a (logical) centralization of the control logic, which relies on standardized APIs, such as OpenFlow [12], to directly program each network device as opposed to using per-device low-level configurations and distributed routing protocols. *SDN controllers* abstract away the complexities of low-level interfaces and allow the network operators to define the network-wide behavior as high-level programs (policies) [13–19]. Due to the asynchronous nature of a network, SDN controllers are event-based systems in which provisioning new policies as well as updating existing ones is communicated via asynchronous protocol over an unreliable network [20]. Meanwhile, the *SDN switches* are simple devices that forward traffic in the network according to the decisions made by the controller. An SDN switch uses a *flow table* (i.e., a look-up table) to forward traffic. Conceptually, SDN switches (in particular, the flow table in each switch) can indeed be seen as memory locations which are *read* and *modified* by various events and entities including the SDN controller. In practice, building such highly asynchronous programs is known to be a very challenging problem, requiring developers to reason about concurrent behavior. As a result, SDN controllers inadvertently introduce harmful-concurrency errors. We demonstrate how these concurrency errors may lead to policy violations or cause downtimes similar to misconfigurations in networks using distributed control planes.

In this dissertation, we produce new methods and techniques for detecting and troubleshooting concurrency violations in production-grade SDN controllers. We start tackling this problem by developing a model that formally captures the ordering of events in an SDN network based on the potential causality relation between them; i.e., happens-before (HB) model. We use this formal model to analyze SDN controllers' events and detect any potential data races in the flow tables' read/write operations. As with standard concurrency analyzers, any SDN concurrency analyzer could report many potential concurrency issues, including false-positive reports. Thus, our second step is to develop techniques to filter out false-positive issues and only report true-positive data races; i.e., harmful violations that can cause the traffic to be forwarded incorrectly or may bring the network down. While detecting data races in flow tables is important, we need to understand these races' broader impact on the network's correctness Thus, our third step is to develop techniques that analyze the reported harmful-concurrency violations and report if of them any caused violations of high-level correctness properties. Finally, we develop techniques to assist the controller developers in troubleshooting and fixing the bugs causing the concurrency violations. In particular, we develop new clustering techniques that use domain-specific knowledge to cluster the harmful violation reports (potentially 1,000s) into a handful of clusters; each cluster represents a bug in the controller. Further, our framework identifies the most representative concurrency violation for each cluster (i.e., those that capture the cause of the violations in the cluster) to assist the SDN controller developers in debugging the root cause of the concurrency issues.

This dissertation is divided into three parts. The first part highlights the necessary background material. The second part focuses on the network-wide configurations synthesis problem. While the third part focuses on detecting and troubleshooting concurrency bugs in SDN controllers.

We begin the second part by motivating the need for network configurations synthesis and defining the network-wide configurations synthesis problem in Chapter 2.

Then, we present the first general network-wide configurations synthesis framework we developed in Chapter 3. In particular, we frame the network-wide configurations synthesis problem as an instance of synthesizing inputs to Datalog program (a declarative logic programming language). We remark that our contributions to input synthesis for stratified Datalog programs extend beyond the domain of computer networks. In Chapter 4, we frame the network-wide configurations synthesis problem as an autocompletion problem. We present techniques that not only allow the network operators to synthesize configurations according to their intents but also allow the network operators to constrain the shape of the output configurations.

We begin the third part by demonstrating the impact of concurrency issues on the correctness and reliability of SDN networks and defining the problem of detecting and troubleshooting such issues in Chapter 5. In Chapter 6, we present a sound and complete dynamic concurrency analyzer for SDN controllers that can ensure a network is free of concurrency violations. We demonstrate how our framework reports harmful violations that could impact the network's correctness and how our framework analyzes the low-level violations to report these violations' impact on high-level correctness properties. Finally, we conclude and suggest future research directions in Chapter 7.

PUBLICATIONS

Most of the work presented in this dissertation appeared previously in peer-reviewed conference and workshop proceedings. The list of accepted publications is presented hereafter.

1. Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever & Martin Vechev. *Net-Complete: Practical Network-Wide Configuration Synthesis with Autocompletion* in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI '18, Renton, WA, USA* (USENIX Association, 2018).
2. Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever & Martin Vechev. *Network-Wide Configuration Synthesis* in *Proceedings of the 29th International Conference on Computer Aided Verification CAV '17* (Springer, Heidelberg, Germany, 2017).
3. Roman May, Ahmed El-Hassany, Laurent Vanbever & Martin Vechev. *BigBug: Practical Concurrency Analysis for SDN* in *Proceedings of the 3rd ACM SIG-COMM Symposium on Software Defined Networking Research SOSR '17* (ACM, Santa Clara, CA, USA, 2017).
4. Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever & Martin Vechev. *SDNRacer: Concurrency Analysis for SDNs* in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '16* (ACM, Santa Barbara, CA, USA, 2016).
5. Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever & Martin Vechev. *SDNRacer: Detecting Concurrency Violations in Software-defined Networks* in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research SOSR '15* (ACM, Santa Clara, CA, USA, 2015).

ACKNOWLEDGMENTS

I look back upon my journey over the many years to reach this point fondly. Despite many challenges and obstacles, I could not be here without the support of my adviser and the great people that I met during this journey. While it would be impossible for me to acknowledge every person who helped me during my Ph.D. journey, many of them deserve special thanks.

First and foremost, I would like to thank Professor Laurent Vanbever for his teaching, patience, encouragement, and guidance over the years. Laurent taught me valuable lessons in identifying research problems and how to be methodological and systemic in solving them. His insightful feedback to my ideas and writings throughout my Ph.D. have been invaluable; his criticisms always constructive; and his logic always clear. I could not have wished for a kinder supervisor. Laurent, thank you for guiding me during this journey! I would like to thank Professor Martin Vechev, who acted as a second adviser, for opening my eyes to the field of programming languages; an area that I always ignored before. Martin taught me valuable lessons in formal methods and opened my eyes to new ideas that I would not understand without his guidance. I would also like to thank Dr. Petar Tsankov for helping me navigate the PL side of research. It has also been a pleasure to work with great co-authors over the years; all of whom have taught me a great deal. Special thanks to co-authors of the papers that appeared in this dissertation: Pavol Bielik, Jérémie Miserez, and Roman May.

I would also like to thank the external reviewers of my dissertation – Professor Aditya Akella, Professor Arvind Krishnamurthy, and Professor Ankit Singla – for taking the time to read and evaluate this work.

I would like to thank my colleagues at the Network Systems Group: Rüdiger Birkner, Tobias Bühler, Edgar Costa Molero, Alexander Dietmüller, Albert Gran Alcoz, Thomas Holterbach, Roland Meier, and Maria Apostolaki for being amazing colleagues. I also thank all my colleagues who have provided valuable feedback on this dissertation's drafts: Rüdiger Birkner, Tobias Bühler, Albert Gran Alcoz, Edgar Costa Molero, and Alexander Dietmüller. Special thanks to Rüdiger Birkner and Tobias Bühler for helping with the German translations of the abstract of this dissertation. Very special thanks to Beat Futterknecht who helped me a great deal throughout my Ph.D. to navigate the Swiss system.

I could not have joined ETH without the continuous support of some great people. I would like to thank Professor Scott Shenker for getting me to believe in research and finishing my Ph.D. again after I almost lost hope in the process. Special thanks to Aurojit Panda for his feedback on my applications and listening to my constant complaints. And I would like to thank Dan Gunter, Brian Tierney, and Inder Monga for helping me to get back on the Ph.D. track at a very critical time of my journey. I also would like to thank Professor Ryan Newton for providing a temporary home for me in his group before joining ETH.

My journey to join ETH passed through Gaza, Delaware, Indiana, and California. At every stop, I met great people who helped me to keep my life balanced in this journey far away from home; despite, sometimes, our short encounters. While I cannot acknowledge every person who touched me on this journey, I would like some of the great friends that without their help, I would not make it this point. I would like Ahmed Radwan for opening my eyes to the research possibilities early on; I will not forget these deep philosophical debates. Special thanks to the fellow residents of Nerd Mansion: Matthew and Debbie Jaffee, Jason Hemann, Miao Zhang, Guilherme Fernandes, and Andrew Younge for being such good companions and for the good memories that I will cherish forever. Special thanks to Jason Hemann for being my remote Ph.D. buddy. Also, I would like to thank Anushka Drescher for her inspiring conversations.

This journey took me far away from my family for over ten years, with almost no way to visit them. Despite the distance, I received constant and unconditional support from them. I would like to thank my family for this unconditional love, who supported me through this really long journey. Mom, Dad, and my seven siblings Dalia, Sami, Alaa, Mohammed, Ali, Samar, and Nasser, thank you all! I also would like to thank Tania Schiesser, who has been the big sister to me in the past few years.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

BACKGROUND

1 INTERNET ROUTING

A computer network is a set of connected devices, called network nodes, exchanging data with each other. Nowadays, network nodes exchange data using the *Internet Protocol (IP)* [21, 22]; a connectionless and unreliable protocol whose goal is to relay datagrams or *IP packets* across one or more networks. *Network traffic* is a group of IP packets moving across the network at a given point in time.

An end-system (the source) generates IP packets then sends them to another end-system (the destination) via the network. Each end-system is assigned a numerical label, called *IP address*, that is 32-bit long in IPv4 [21] or 128-bit long in IPv6 [22]. When an end-system generates a packet, it labels it with the *source IP address* (the address of the end-system which generated the packet) and the *destination IP address* (the address of the end-system to whom this packet is destined to). Related end-systems (e.g., all students or all staff members) are usually grouped and addressed together using the concept of an *IP prefix*. An IP prefix (often denoted as `IP/n`) identifies a set of IP addresses whose first `n` bits are identical. These `n` bits identify the "network" part of the IP address, while the remaining bits often identify the end-system part of the address.

In this chapter, we introduce the main concepts of how IP networks work. First, we present the IP router in Section 1.1; which is the primary device that is in charge of computing in-network paths and forwarding traffic. Then, we briefly present, in Section 1.2, the different paradigms of how networks compute paths along which to forward traffic. The following three sections, sections 1.3 to 1.5, highlight how inter-domain routing, intra-domain routing, and Software-Defined Networking (SDN) implement IP routing. Finally, we define network policies in Section 1.6.

1.1 IP ROUTER

End-systems belonging to the same IP prefix can communicate with each other directly or via devices such as *network switches*, while end-systems belonging to different prefixes have to communicate via network gateways known as *IP routers*. IP routers are composed of two logical planes: the *control plane* and the *data plane*; see Figure 1.1. The control plane decides how to forward packets (the path(s) each packet takes in the network to reach its destination), while the data plane forwards the traffic accordingly.

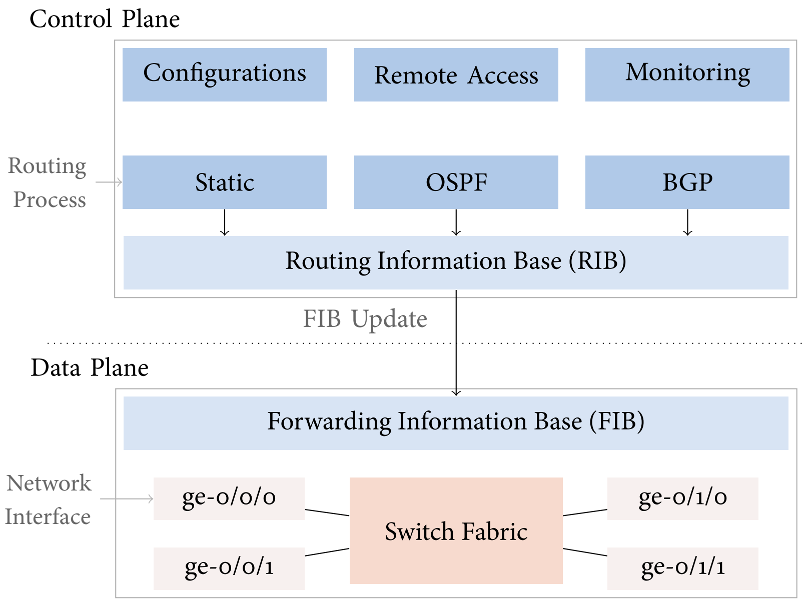


FIGURE 1.1: The high-level architecture of an IP Router.

In the control plane, *IP routing* is the process of computing *routing paths* to destination IP prefixes in the network. A routing path is a set of network nodes that a packet traverses to reach its destination. A router runs one or more routing protocols to execute the IP routing task. Routing protocols compute the in-network routing paths based on per-router local configurations, defined by the network operators, and the information exchanged with other routers in the network (called *peers* or *neighbors*). Each router stores the computed paths in a designated table called the *Routing Information Base (RIB)* or the *routing table*.

By adapting the configuration of each device, network operators can control their routing behavior (e.g., which paths they prefer) and, by extension, their forwarding behavior. To forward traffic, each router stores for each destination the *next hop* (i.e., a neighbor router that can reach the destination) in a designated table called the *Forwarding Information Base (FIB)* or the *forwarding table*. The union of all the forwarding tables in the network is the *network-wide forwarding state*. Note that the forwarding table is a subset of the routing table. The network operators configure, as local per-router configurations, how a router selects entries from the routing table as forwarding table entries.

In the data plane, *packet forwarding* is the act of sending packets along the selected forwarding paths to the destination of each packet (the destination IP address in the packet's header).

1.2 IP ROUTING

Traditionally, routers perform the IP routing task using one or more *distributed routing protocols*. When using distributed routing protocols, each router runs an instance of the protocol (in its control plane) and exchanges information (via protocol-specific messages) about the network with the neighboring routers running the same protocol. Then each router, locally, uses the information it learned about the network and local parameters (i.e., per-router *configurations* defined by the network operator) to compute the routing paths in the network.

A *routing domain* is a group of network routers managed by a single administrative entity (e.g., one organization or team of network operators) [23]. In the context of the Internet, an *Autonomous System (AS)* is a routing domain. We remark that network operators may partition their network into multiple ASes to better enforce routing requirements within the network. We refer to the internal ASes under the operators' control as *private* and to the ASes visible on the Internet as *public*.

Nowadays, there are two families of distributed routing protocols: *intra-domain* and *inter-domain*. Intra-domain routing protocols compute routing paths within one routing domain, while inter-domain routing protocols compute routing paths across multiple routing domains.

Routing protocols differ by the type of routing paths that they are capable of computing. For instance, routing protocols based on the Dijkstra algorithm (e.g., the Open Shortest Path First protocol (OSPF)) can only compute routing paths that direct traffic along least-cost-paths. While policy-based protocols (e.g., the Border Gateway Protocol (BGP)) can direct traffic along non-shortest paths. In the meantime, OSPF supports multi-path routing and is thus better suited for load-balancing traffic (a feature heavily used in practice), while BGP cannot forward traffic along multiple paths by default.

1.3 INTRA-DOMAIN ROUTING

Intra-domain routing is concerned with computing routing paths within a single administrative domain. Intra-domain protocols, also called *Interior Gateway Protocols (IGPs)*, provide optimal routing based on given criteria; e.g., link capacity or the number of hops.

There are two families of intra-domain routing protocols: *distance-vector* and *link-state* protocols.

Distance-vector routing protocols. In this family, the distributed routing protocol computes the best route based on the *distance*; i.e., a numerical metric (e.g., the number of hops) to estimate the cost to reach the destination via this path. Generally, this family of routing protocols works as follows: each router periodically exchanges, with all its neighbors, the *routing table* (i.e., a table that lists network destinations and how to reach them) and a distance to each destination in the network. Then, upon receiving the updated routing table from its neighbor, each router updates the distance of each destination and uses a distance-vector algorithm (e.g., Bellman-Ford algorithm [24–26]) to compute the best route to each destination. This family includes routing protocols such as RIP [27], EIGRP [28], DSDV [29] and Babel [30].

Link-state routing protocols. Unlike the distance-vector routing protocols, the link-state routing protocols do not share the distance vector to each destination. Instead, each router advertises the state of each of its links to every other router. Then each router builds an IGP adjacency graph (i.e., a weighted graph that represents the connectivity of the network) and uses it to compute the routing table. Routers construct the IGP adjacency graph in three steps. First, each router in the network discovers its direct neighbors via periodically sending HELLO messages over all its connected links. Second, each router floods the network with a *link-state advertisement (LSA)* containing: *(i)* the ID of the router that created the LSA; *(ii)* cost of links to reach its directly connected neighbors; *(iii)* a sequence number identifying the LSA version; and *(iv)* and a time-to-live (TTL) value for the LSA to prevent it from infinitely looping the network. Finally, every router constructs and IGP adjacency map of the network based on the received LSAs and compute the best route(s) to each destination in the network; generally, using a variant of the Dijkstra algorithm [31]. This family includes protocols such as Open Shortest Path First (OSPF) [32, 33] and Intermediate System to Intermediate System (IS-IS) [34].

1.4 INTER-DOMAIN ROUTING

Inter-domain routing is concerned with computing routing paths between different administrative domains. Path-vector protocols are commonly used for this purpose. Generally, path-vector protocols work as follow: each router periodically exchanges, with its neighbors, a path vector (i.e., a list of routers or ASes) to reach each network destination. Then, each router uses a local policy to select the best route to use to forward traffic; for instance, the local policy might select the best route based on the business agreement between two neighbors.

Name	Description
`Prefix`	A value that represents a set of destination IP addresses
`LocalPref`	A positive integer that indicates the degree of preference for one route over the other routes
`Origin`	The origin of the announcement: `IGP`, `EGP`, or `Incomplete`
`MED`	(Multi-Exit Discriminator) A positive integer that indicates which of the multiple routes received from the same AS is preferred
`ASPath`	The AS path to reach the destination, we use `ASPathLen` to denote the length of the AS path
`NextHop`	The router to which to forward packets
`Communities`	A list of tags carried with the announcement

FIGURE 1.2: Commonly used attributes in BGP announcements.

In today's Internet, the Border Gateway Protocol (BGP), a path-vector protocol, is the only used inter-domain routing protocol [35]. The network operators configure each BGP-enabled router to establish *peering sessions* (a TCP connection) with other BGP-enabled routers. The BGP routers exchange routing information (i.e., BGP *announcements*) via the peering sessions. There are two types of peering sessions; *eBGP* and *iBGP*. eBGP peering sessions connect routers in different ASes, while iBGP peering sessions connect routers within a single AS.

Network operators may configure each BGP router with custom import and export filters; i.e., *the BGP policies*. A router processes each received announcement using the import filters, which may drop the announcement or modify its attributes; see Figure 1.2 for the commonly used attributes. Then, the router selects one announcement (of possibly many) for each IP prefix as the *best route*. Finally, the router processes best-routes using the export filters, which may drop the announcement or modify its attributes, then it forwards the result to its neighboring routers. Note that a router propagates announcement it learns from its neighbors *if and only if* the router selects the announcement as the best route, and the router's BGP export filters do not drop the announcement.

BGP routers use a tie-breaking process when receiving (after applying the import filters) multiple announcements for the same destination prefix. This tie-breaking process works in multiple steps until the process breaks the tie and selects only one announcement as the best route. The following is the standard tie-breaking process[1] [35]:

1. Prefer higher `LocalPref`,
2. Prefer shorter `ASPathLen`,
3. Prefer lower origin type: `IGP` < `EGP` < `Incomplete`,
4. Prefer lower `MED` for announcements learned for the same peering AS,
5. Prefer announcements from external routers;
6. Prefer lower `IGPCost`, calculated by the network's Internal Gateway Protocol (IGP), such as OSPF,
7. Prefer lower BGP identifier value (an unsigned integer value configured for each BGP router),
8. Prefer the route received from the lowest peer address.

1.5 SOFTWARE-DEFINED NETWORKING (SDN)

While distributed routing protocols have many advantages (e.g., scalability, robustness), the physical coupling of the two logical planes in one device (the IP router) has limited the innovation in computer networks. Commercial vendors traditionally provide IP routers as closed boxes with limited access for researchers and third parties [12]. Recently, *Software-Defined Networking (SDNs)* has emerged as an alternative paradigm to distributed routing protocols that couples the control and data planes into IP routers. At its core, SDN is predicated around two key principles. First, SDN argues for a physical separation between the control plane and the data plane. Second, SDN argues for a (logical) centralization of the control logic, which relies on standardized APIs, such as OpenFlow [12], to program forwarding state in each network device (SDN switch).

In SDN, the network consists of multiple SDN-enabled switches and a logically-centralized controller. The switches communicate with the controller via standardized APIs such as OpenFlow [12]. Each switch forwards packets based on the content of its forwarding table. The forwarding table of a switch is an ordered (by *priority*) list of forwarding entries composed, among other things, of a *match*, *actions*, and (optionally) a *timeout* value. The match is a boolean predicate that identifies a set

1 Some vendors extend the standard BGP tie-breaking process with additional steps, or give the option to disable some of the steps.

of packets to which the corresponding forwarding actions are applied. Forwarding actions include modifying the packet's header (e.g., change the destination IP address) or sending the packet to the controller or a given output port. There are two types of forwarding entry timeout values set by the controller: *soft-timeout* or *hard-timeout*. Soft-timeout value purges the forwarding entry after a set time since the last packet matched it. Hard-timeout value purges the forwarding entry after a fixed amount of time has elapsed since the controller wrote the entry to the flow table. Forwarding entry timeouts usually trigger a subsequent notification to the controller.

The SDN controller is in charge of computing the forwarding state and populating each switch's forwarding table with the information necessary to forward packets; this process is called a *network update*. A network update is triggered either *proactively* or *reactively*. External events to the network such a manual change from the network operator trigger proactive network updates, while internal network events such as link failures trigger reactive network updates.

1.6 NETWORK POLICIES

Although the primary job of a computer network is to deliver IP packets via the data plane, networks usually have many additional requirements on which packets to deliver and how to deliver them. We refer to the set of requirements specified by the network operators as the *network policies* or *intents*. More precisely, network policies are a set of conditions and constraints that specify: *(i)* which packets the network must deliver or which packets the network must drop (e.g., for security reasons); and *(ii)* what path(s) the network should use to deliver each packet. In the context of network policies, *traffic class* is a set of IP prefixes handled according to the same policy. Network operators often derive the network policies from business, performance, and security objectives. For instance, the network operators of an Internet service provider (ISP) might define the policies of how their network forwards traffic other networks based on their business peering agreements [36].

Traditionally, the high-level network policies are defined informally, and often scattered across human-readable documents (e.g., network design, requirements, and specifications documents). This informal approach created ambiguity about what the network should do and made it harder to automate network management. In recent years, researchers developed formal languages to capture high-level policies [13, 14, 16, 17, 37–44]. These policy languages provide the network operators valuable tools to unambiguously express the expected behavior of the network precisely and concisely.

Implementing network policies is the act of translating the high-level policies into a *compliant forwarding state*; i.e., a forwarding state that forwards packets according to the intended policies. Implementing network policies differs widely based on the network's type.

In SDN networks, the controller directly compiles the high-level policy into a compliant forwarding state. For instance, ONOS, a production-grade SDN controller, provides an intent framework that allows the operators to express their intents (policy) in high-level languages. Then ONOS automatically compiles a compliant forwarding state and pushes it to switches [45].

On the other hand, implementing policies in traditional networks is not as direct as SDN networks. In traditional networks, the network operators translate the high-level policies to low-level configurations; low-level configurations are often vendor-dependent. Note that the network operators have to derive low-level configurations for every device in the network. For brevity, we say that configurations *implement* a routing policy, while, in fact, the distributed routing protocol uses the given configurations as an input to compute a forwarding state.

Part II

NETWORK CONFIGURATIONS SYNTHESIS

2 NETWORK CONFIGURATIONS SYNTHESIS PROBLEM STATEMENT

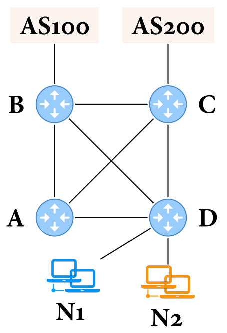
Today's computer networks are large dynamic distributed systems whose requirements keep evolving with the business needs. Thus, the network operators keep changing the network policies to meet the ever-evolving requirements. To implement network policies, network operators translate the high-level policies into low-level, often vendor-dependent, configurations for every device in the network; see Section 1.6.

However, translating the high-level policies into *correct configurations* is very challenging and is prone to errors. The network operators have to understand potentially thousands of configuration files and complex-distributed protocols, then decide on possible changes to parameters scattered across the configuration files and predict the impact of these changes to the computed forwarding state. Errors in translating the high-level policies into configurations lead to misconfigurations. More precisely, *misconfigurations* are configurations that incorrectly implement the intended high-level policies. Misconfigurations violate the intended policies and can cause network downtime. Ultimately, misconfigurations can lead to substantial financial and reputational losses. Every few months, downtimes involving major players such as Verizon [2, 46], United Airlines [6], Google [4, 7, 47], NYSE [8], or Facebook [1] make the news.

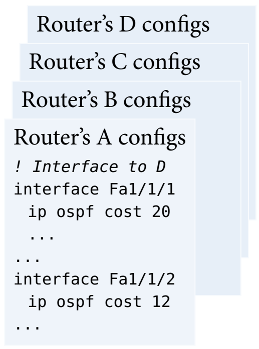
Several studies show that human-induced misconfigurations, *not* physical failures, explain the majority of downtimes [3, 10, 11]. To reduce misconfigurations due to human errors, network operators would write the network-wide policies as high-level *declarative specifications*. The high-level declarative specifications describe the forwarding state they desire rather than the details of how the routing protocols would converge to that state. Then they use a tool that automatically translates the network policies into correct low-level configurations; we refer to this step as *network-wide configurations synthesis*.

We start this chapter with a motivating example of how network operators manually configure a network highlighting the key challenges and pitfalls in Section 2.1. Then, we formally define the *network-wide configurations synthesis problem* and present the design space and the challenges for designing network configurations synthesis frameworks in Section 2.2. In Section 2.3, we highlight our main contributions to address this problem. Finally, we review the related work of the network-wide configurations synthesis problem in Section 2.4.

2.1 MOTIVATING EXAMPLE



(a) Network Topology T


Router's D configs
Router's C configs
Router's B configs
Router's A configs

```
! Interface to D
interface Fa1/1/1
  ip ospf cost 20
  ...
...
interface Fa1/1/2
  ip ospf cost 12
...
```

(b) Configurations $\mathcal{C}_{old}$

1. Traffic from A to $N1$ prefers: $A \rightarrow C \rightarrow D$.
2. Traffic from A to $N2$ prefers: $A \rightarrow D$.
3. Traffic from A to Ext prefers: $A \rightarrow B \rightarrow AS100$.
4. Traffic from D to Ext prefers: $D \rightarrow C \rightarrow AS200$.

(c) New Policy $\varphi_{R_{new}}$

FIGURE 2.1: To configure a network, the network operators consider the network topology (2.1a), any existing configurations (2.1b) and the new routing policy to implement (2.1c) as inputs.

In this section, we present a motivating example to showcase the need for network-wide configurations synthesis. Specifically, we present a typical workflow of manually configuring a network to implement new routing policies. We remark that the exact workflow of configuring a network varies from one network to another. Here, we focus on a generic workflow to highlight the key challenges in manually configuring a network in order to motivate the need for automation.

In our example, the network operators are given new routing policies to implement on an existing network; see Figure 2.1. The new policies might reflect security policies or might be generated by a traffic engineering optimization tool [48, 49]. Furthermore, the network operators may formally specify the new policy using a network policy language; see Chapter 1. Additionally, we assume that existing routing policies (implemented by the existing configurations) are known to the network operators, e.g., via documentation or tools such as Net2Text [50]. The task of the network operators is to change the existing configurations and produce new configurations compliant with the new policy; see Figure 2.2.

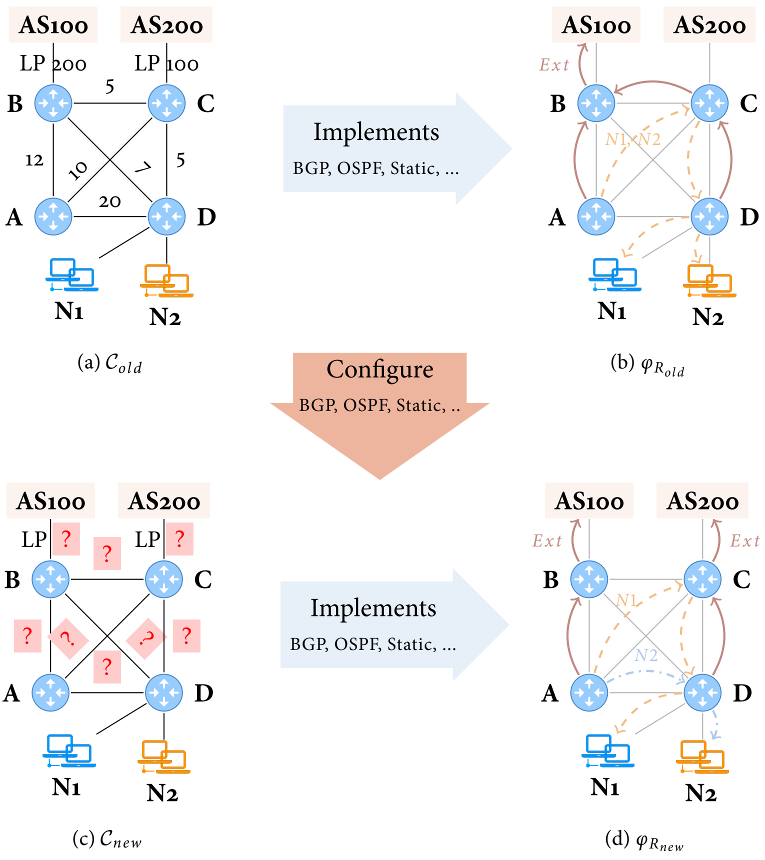


FIGURE 2.2: Network operators are presented with a network that is configured with $\mathcal{C}_{old}$ (2.1b) and implements a known policy $\varphi_{R_{old}}$ (2.2b). The goal of the network operators is to find a new configuration $\mathcal{C}_{new}$ that implements the required new policy $\varphi_{R_{new}}$.

2.1.1 *Inputs*

Now, we start describing the inputs that network operators typically consider before configuring a network; see Figure 2.1.

Input* 1*: Network Topology* T*. We consider a simple network topology, depicted in Figure 2.1a, composed of 4 routers denoted A, B, C and D. Routers B and C connect to two provider ASes; $AS100$ and $AS200$, respectively. Router D connects two internal subnets `N1` and `N2`.

Input* 2*: Existing Configurations* $\mathcal{C}_{old}$ *and Policy* $\varphi_{R_{old}}$*. The existing routing policies $\varphi_{R_{old}}$ are as follow; see Figure 2.2b. For OSPF, the network prefers forwarding traffic entering the network at router A with a destination to either $N1$ or $N2$ along the path $A \rightarrow C \rightarrow D$. While for BGP, $AS100$ is the preferred provider to reach external prefixes; hence, each router uses the least-cost OSPF path to router B that is connected to $AS100$.

The network is running OSPF and BGP routing protocols. The existing configurations $\mathcal{C}_{old}$ are as follow. An iBGP mesh is running on all the routers in the network. Additionally, the network is configured to peer with two ASes ($AS100$ and $AS200$) to reach external prefixes `Ext`. The BGP configurations set the `LocalPref` to 100 for the prefixes that router C learns from $AS200$, while it sets the `LocalPref` to 200 for all prefixes that router B learns from $AS100$; making $AS100$ the most preferred peer. For OSPF, we mark the cost of each link on the topology in Figure 2.2a. We assume the link cost is symmetrical; i.e., the cost is the same in each direction.

Input* 3*: New Routing Policy* $\varphi_{R_{new}}$*. In this example, the network operators want to implement new internal and peering routing policies $\varphi_{R_{new}}$; see Figure 2.2d.

The new internal routing policy load-balances traffic entering the network through router A as follows. The network should prefer forwarding traffic along the path $A \rightarrow C \rightarrow D$ if the destination of the traffic is $N1$. While if the destination of the traffic is $N2$, then the network should prefer forwarding the traffic along the direct path $A \rightarrow D$. We remark that this policy only defines the first preference and other available paths should be used in case the most-preferred path is not available, hence guaranteeing reachability.

If traffic with a destination `Ext` is entering the network at router A, then the network should prefer forwarding that traffic along the path $A \rightarrow B \rightarrow AS100$. While if the traffic with a destination `Ext` is entering the network at router D, then it should exit via router C to $AS200$.

2.1.2 *Configurations Pitfalls*

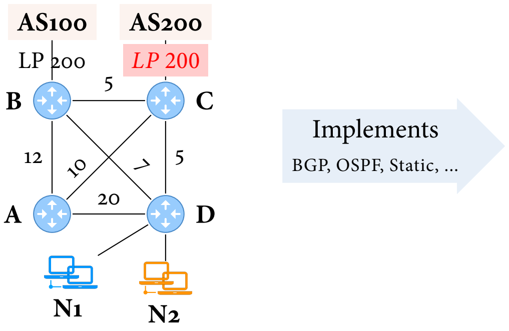
We show some of the common misconfigurations pitfalls that network operators might encounter while manually configuring a network.

Pitfall 1: Not Considering Protocol Dependencies. Let us consider changing the BGP import policies to comply with $\varphi_{R_{new}}$. Intuitively, the new policy requires using both ASes ($AS100$ and $AS200$) as upstream providers. Considering $\mathcal{C}_{old}$ (Figure 2.2a), one standing observation is that the `LocalPref` values are not equal in the BGP import policies; hence, BGP selects only the AS with the highest `LocalPref` value. One easy change is to reconfigure BGP import policies such that prefixes learned from either $AS100$ or $AS200$ have the same `LocalPref` values; see Figure 2.3a. However, this solution is *not sufficient* to implement $\varphi_{R_{new}}$ because it does not consider the IGP cost. By just changing the `LocalPref`, iBGP is going to propagate routes from $AS100$ and $AS200$ to router A. Router A is going to prefer routes learned from $AS200$ via path $A \rightarrow C$ because the path $A \rightarrow C \rightarrow AS200$ has a lower OSPF cost than the path $A \rightarrow B \rightarrow AS100$ [1]; see Figure 2.3b. Such misconfiguration could have catastrophic implications. For instance, this misconfiguration could congest the link $A \rightarrow C$ (i.e., the network sending more traffic over this link than its capacity) or even worse sending traffic using the link $A \rightarrow C$ might break some security policies; some security policies might prevent using less secure links for specific traffic classes.

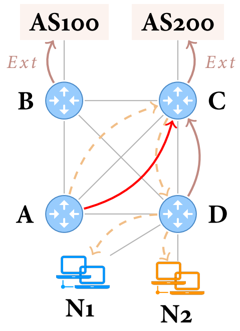
Pitfall 2: Only Considering Local Values. To implement the new BGP policy in $\varphi_{R_{new}}$, the network operators have to change both the BGP import policies and OSPF link weights. After changing the `LocalPref`, one possible change to IGP paths is to lower the link cost from A to B. However, if the network operators set the link cost to a low value, see Figure 2.3c, traffic destined to $N1$ and $N2$ will start flowing through the path $A \rightarrow B \rightarrow C \rightarrow D$; see Figure 2.3b. In this example, the network operators need to consider weights such that the cost of path $A \rightarrow B$ is less than the cost of path $A \rightarrow C$ but greater than the cost of the path $A \rightarrow C \rightarrow D$.

Pitfall 3: Protocols Expressiveness. Networks run one or more routing protocols that are capable of expressing different policies; i.e., different protocols are capable of computing different forwarding paths. The network operators need to consider if the protocols running in the network are capable of implementing the intended policies or not. For instance, in OSPF, the traffic for all prefixes between any two routers must follow the same path (or paths in the case of ECMP). Hence, OSPF is not capable of implementing the internal routing requirements in $\varphi_{R_{new}}$; since it requires using different paths between routers A and D to forward traffic destined for $N1$ and $N2$. In this case, the network operator must use a different protocol to implement this

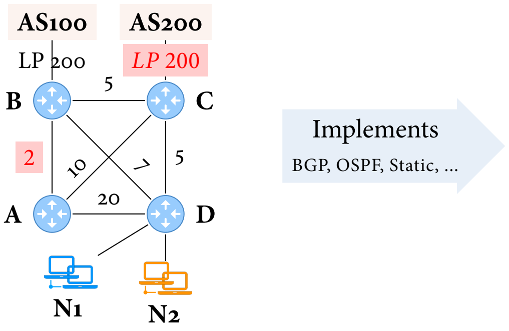
1 For simplicity, we assume that the routers are configured to ignore AS path attribute.



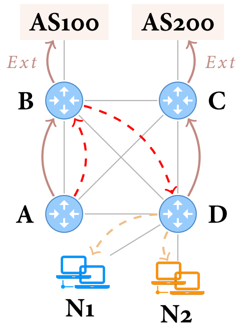
(a) C_{wrong1}



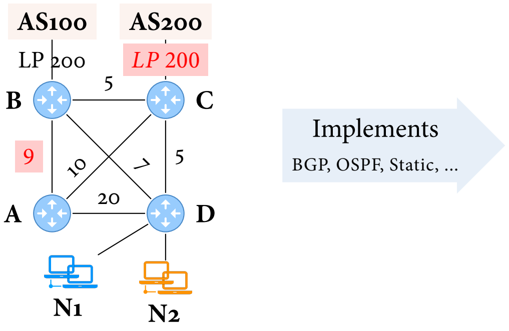
(b) $\varphi_{Rwrong1}$



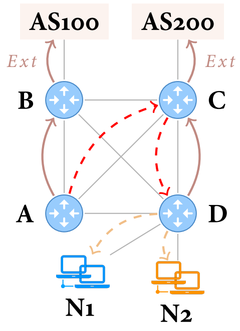
(c) C_{wrong2}



(d) $\varphi_{Rwrong2}$



(e) C_{wrong3}



(f) $\varphi_{Rwrong3}$

FIGURE 2.3: Common misconfigurations pitfalls. Figures 2.3a and 2.3b show a misconfiguration due to not considering the dependencies between routing protocols. Figures 2.3c and 2.3d show a misconfiguration that is caused by only considering local link costs. Figures 2.3c and 2.3d show that the routing policy cannot be implementable by OSPF.

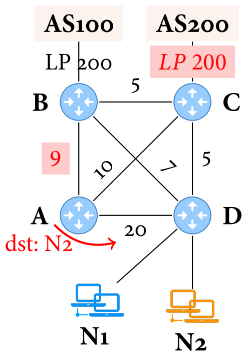


FIGURE 2.4: Correct configuratoins C_{new} that implements the routing policies $\varphi_{R_{new}}$ in Figure 2.1c.

requirement. For instance, the network operators could keep the existing OSPF link costs such that traffic destined to $N1$ uses the path $A \rightarrow C \rightarrow D$, but install a static route at A to forward traffic destined to $N2$ via the direct path $A \rightarrow D$.

2.1.3 *Correct Configurations*

Now, we present a correct configuration C_{new} that implements the routing policies for our example. We remark that there are multiple configurations to implement the given routing policies; we only show one of them. The changes to the old configurations are as follow:

1. Set the `LocalPref` for announcements learned from $AS200$ to 200.
2. Set the link weight for the link $A - B$ to 9.
3. Setup a static route on router A such that traffic destined to prefix $N2$ goes directly to router D via the link $A - D$.

2.2 THE NETWORK-WIDE CONFIGURATIONS SYNTHESIS PROBLEM

Informally, a network-wide configurations synthesis framework takes as inputs (see Figure 2.5): *(i)* a model of the network (also called *network specification*); *(ii)* any existing configurations; *(iii)* the network topology; and *(iv)* high-level routing policies. The network specifications capture the behavior of the various distributed routing protocols running in the network and the interactions between them, allowing

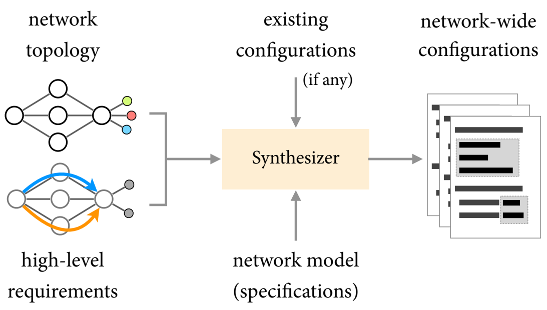


FIGURE 2.5: A network-wide configurations synthesis framework takes as input: network topology, routing policies, any existing configurations, and a network model of the routing protocols and the interactions between them, then produces network-wide configurations that induce a forwarding state satisfying the intended policies.

the synthesizer to predict the forwarding state of the network under given inputs. Network operators often configure networks incrementally as the business needs change. If a synthesizer accepts the existing configurations as input, the synthesizer will be able to produce configurations with minimal changes and are similar in style to the existing ones. The network topology defines the devices in the network and the links between them. The high-level policies capture the network operators' intents. Formally, we define the network-wide configurations synthesis problem as follows:

Definition 2.2.1. *Network-wide Configurations Synthesis Problem*
Given a network specification $\mathcal{N}$, which defines the behavior of all routing protocols run by the routers, a set of routing policies φ_R defined over the network-wide forwarding state, and (optionally) any existing configurations $\mathcal{C}_{old}$, find a new configurations $\mathcal{C}_{new}$ such that the routers converge to a network-wide forwarding state satisfying φ_R.

2.2.1 *Configurations Synthesis Design Space*

To solve the network-wide configurations synthesis problem, one must ask three essential questions: *(i)* which routing protocols to support?; *(ii)* how to deal with existing configurations?; and *(iii)* how to build synthesis frameworks that can scale to support large production networks?

Supporting Multiple Routing Protocols

Network operators prefer to rely as much as possible on distributed routing protocols (as opposed to using static routes) to compute the forwarding state to ensure network reliability and scalability. For instance, a typical ISP runs IS-IS as the IGP protocol and uses BGP to peer with other ISPs. While warehouse-scale data centers often choose to run a single routing protocol (e.g., BGP) to simplify its operations.

A network-wide configurations synthesis framework might support one or more routing protocols. The choice of how many and which routing protocols a given network-wide configurations synthesis framework supports affects the *practicality*, *expressivity*, and *scalability* of the system. For instance, if a synthesis framework supports a single routing protocol, it is only applicable to networks running one protocol (e.g., data centers running only BGP). Additionally, these frameworks can only support the policies that the given protocol can implement. While using more routing protocols may allow the network operator to implement richer policies. A practical configurations synthesizer should strike a balance between the number of protocols running in the network and classes of network policies it can support. This balance should maximize the use of each routing protocol and avoid the complexities of running unnecessary routing protocols. Ideally, the network operator should be able to define a preference over which routing protocols to be used to implement a given policy.

Handling Existing Configurations

A network-wide configurations synthesis framework can choose to either generate new configurations from scratch for every policy change or generate minimum changes to the existing configurations.

Generating new configurations from scratch, whenever the policy or the network topology changes, enables the synthesis framework to choose a configurations style that is optimal for synthesis. While this choice allows for greater scalability, operators rarely configure networks from scratch in practice; instead, they implement small incremental policy changes during the lifetime of the network. Additionally, generating entirely new configurations for every policy change creates additional operational overhead. Namely, this approach creates complexities in configurations management and potentially increases the convergence time of routing protocols; as more routers are affected by the change.

A general synthesis framework should be able to accept the initial configurations and to synthesize new configurations that only reflect the policy change; i.e., the changes in the configurations should be proportional to the changes in the policy. In general, the configurations generated and the subsequent changes should be small in size such that a human operator or static analyzers can easily verify them.

Scalability

A practical configurations synthesis framework should be able to scale to handle production-size networks in terms of the number of routers in the network and the number of traffic classes defined in the policy. As we described, the framework's choice of the supported routing protocols and the style of the output configurations affects the scalability of the system.

Distinct routing protocols often depend on one another, making it challenging to ensure that they collectively compute a compatible forwarding state. For instance, BGP uses the path costs computed by the intra-domain protocol (e.g., OSPF) to select the best route. This dependency between protocols complicates the synthesis process; the synthesizer, in some cases, cannot simply synthesize configurations for each routing protocol independently, but rather need to consider them jointly. Synthesis frameworks designed to support only one routing protocol can leverage many protocol-specific optimizations without having to worry about interactions between different routing protocols. This design choice enables such frameworks to synthesize configurations for large network sizes (>1, 000 devices).

Regardless of how the synthesis framework handles any existing configurations, the search space of configuration parameters is massive, and it is thus difficult to find configurations that implement the intended policies. To make any synthesis framework feasible to use, the synthesis framework needs to utilize intelligent search heuristics to prune the search space of the possible configurations.

2.3 OUR CONTRIBUTIONS

We next summarize our main contributions to address the network-wide configurations synthesis problem.

1. In Chapter 3, we present a general solution to the network-wide configurations synthesis problem. In particular, we formulate the network-wide configurations synthesis problem in terms of synthesizing inputs to a stratified Datalog program. The stratified Datalog program captures the behavior of the network; i.e., the distributed protocols ran by routers together with any protocol dependencies. Here, the output (the fixed point) of a Datalog program represents the stable forwarding state of the network. Then we develop an algorithm to find an input for a stratified Datalog program where the program's fixed point satisfies the given requirements. This new approach allows us to

build a general network configurations synthesis framework (called `SyNET`); i.e., it can synthesize configurations for any routing protocol that can be expressed as stratified Datalog. However, as we show in our evaluation, the generality comes at the cost of scalability. We note that this contribution solves the general stratified Datalog input synthesis problem.

2. In Chapter 4, we describe a sketch-based configuration synthesis approach and the `NetComplete` system based on it. In this approach, the network operators express the high-level policies in addition to constraints on the output configurations. The constraints on the output configurations are expressed by sketching parts of the existing configuration that should remain intact (capturing a high-level insight) and "holes" represented with symbolic values which the synthesizer should instantiate (e.g., OSPF weights, BGP import/export policies). `NetComplete` then autocompletes these "holes" such that the resulting configuration leads to a network that exhibits the required behavior. This new approach allows us to build a system that supports multiple routing protocols and yet scales to larger network sizes while supporting existing configurations and producing changes that are understandable by network operators.

2.4 RELATED WORK

Intent-based Networking and SDN Languages. The importance of relying on high-level abstractions in network management has received considerable attention, specifically in the context of Software-Defined Networking (SDN) [14, 16, 38, 39, 41–43, 51, 52]. This influence goes beyond academics with two of the largest production-grade SDN controllers (ONOS and OpenDayLight) now providing declarative network management [45, 53].

Our work brings programmability to traditional networks, by enabling operators to enforce policies expressed in high-level SDN-like languages such as Genesis [38] or Frenetic [14]. Our work, therefore, complements the above initiatives and enables them to be used beyond OpenFlow or P4-enabled networks.

Network Configuration Synthesis. Recently, multiple works have aimed at synthesizing configurations out of high-level requirements [37, 54–58].

ConfigAssure [54] is a general system that accepts requirements in first-order constraints as inputs and outputs a configuration satisfying the requirements. For example, ConfigAssure can be used to express requirements such as "all interfaces must have unique addresses," and it can automatically compute a compliant configuration that assigns addresses to interfaces. The fixed point computation performed by routing protocols cannot be captured using the formalism used in ConfigAssure. Additionally, as shown in our evaluation for `NetComplete` and `SyNET`, a direct encoding of routing computations into constraints goes beyond what existing solvers can handle. Therefore, ConfigAssure cannot be used to specify networks' routing policies and, in turn, to synthesize protocol configurations for networks.

Route Shepherd [57, 58] takes a partial specification of BGP preferences and derives constraints over link costs that capture the absence of BGP instability. In contrast, our work models the derivation of BGP preferences and also synthesizes the BGP configuration.

Propane [37] and PropaneAT [55] produce BGP configurations out of high-level requirements. Having the freedom to output any configuration enables these systems to use templates and, in turn, to scale to large networks. In contrast, both techniques we develop in this dissertation support partial configurations for multiple protocols (OSPF, BGP, and static routes), which prevents us from leveraging specific protocol-specific templates. While techniques pay for this flexibility in terms of scalability, they are still fast, synthesizing configurations for practical network-sizes within a reasonable time (seconds for `NetComplete` and less than 24h for `SyNET`).

Zeppelin [56] synthesizes configurations for traditional control planes (i.e., using BGP, OSPF, and static routes). Zeppelin uses a two-phase approach. First, Zeppelin uses Genesis [38] to synthesize a set of forwarding paths compliant with the given policy (note, there are zero or more sets of forwarding paths compliant with a given policy). Second, Zeppelin tries to synthesize control-plane configurations such that the control plane converges to the synthesized forwarding paths in the first step. If Zeppelin fails to find configurations to implement the given forwarding paths, it retries again using Genesis to generate a new set of policy-complaint forwarding paths. In our work, we assume a front-end that accepts SDN-like policy language (e.g., Genesis [38], Propane [37]) but we did not implement a full loop to retry with different forwarding paths as in Zeppelin. Zeppelin encodes the notation of hierarchical control planes; e.g., it groups the routers into domains running OSPF and connects them via BGP and static routes. Zeppelin uses a greedy stochastic search algorithm, Markov Chain Monte Carlo (MCMC) sampling method [59], to find a domain assignment that minimizes BGP configurations overhead (i.e., the number of local preference entries) and the need for static routes. While `SyNET` has no notation of domains, `NetComplete` allows the operator to encode the OSPF domains directly in the configurations sketch. The domain assignment can be done manually or automatically using the Zeppelin engine. To synthesis OSPF configurations, Zeppelin uses an LP-solver making it 2-3 orders of magnitude faster than the direct SMT encoding used in `SyNET`. However, the CEGIS algorithm we developed in `NetComplete` achieves similar speedups. Moreover, Zeppelin directly encodes the notation of resiliency (only, 1-link failures). In our work, we encode resiliency as any-path requirements. In which, the operator provides multiple paths as a requirement allowing, potentially, support for more than 1-link failure. Zeppelin gives the network operators coarse-grained control over the output configurations (e.g., the number of domains), while `NetComplete` gives the operator fine-grained control of the synthesized configurations.

CPR [60] uses high-level abstraction to represent the network's control plane as graphs. Then it poses the synthesis problem as finding the minimal changes on the computed graphs to reflect the new policy. This problem formulation allows CPR to be very efficient in implementing small policy changes on existing configurations; however, unlike `SyNET`, it is not able to synthesize configurations from scratch. CPR can only synthesize configurations that are slightly different than the input configurations, while in `NetComplete`, the network operator can use both fine-grained and coarse-grained configuration sketches allowing `NetComplete` to synthesize configurations for policies that are widely different.

Synthesizers such as NetEgg [40, 61], NetGen [62], and Genesis [38] target SDN environments and aim to derive controller programs (instead of configurations) out of requirements. While their goal is similar to ours, our target is different (distributed protocols vs. centralized controller).

Network Verification. Network verification approaches are used to check if the configurations correctly implement the high-level policies. In contrast, our work in `NetComplete` and `SyNET` focuses on synthesizing correct configurations, which subsumes verification; e.g., `NetComplete` can verify policies given the configuration sketch is concrete (with no holes).

For networks running distributed routing protocols, researchers introduced two main approaches to check if a given configuration is correct. The first approach is to use static analysis tools to verify the correctness of the configuration files [63–66]. However, the existing configuration analysis tools handle configurations complexity by restricting the scope of application; i.e., considering only specific protocols or network properties. The second approach is to model the entire control plane [64, 67–72]. In this approach, the tools designers have to make a trade-off regarding the fidelity of the control-plane model; i.e., the level of details included in the model. Using a high-fidelity model (i.e., modeling everything) results in a slow verification tool [67]. In contrast, using a more abstract model affects the accuracy of the model and may prevent the network operator from checking the correctness of certain properties [64, 68]. Further, to make use of such tools, the network operators need to provide the intended high-level policy as invariants over the network-wide forwarding state. Next, we summarize the closely related verification tools.

FSR [72] encodes BGP preferences using routing algebra [58] and verifies safety properties (e.g., BGP stability) using SMT solvers. Bagpipe [69] formalizes BGP and presents an analyzer for BGP configurations. While our work focuses on the multi-protocol case.

ARC [68] uses a novel graph-based high-level abstraction to verify the correctness of a network's control plane under arbitrary link failures. The key to ARC's scalability is that ARC can verify a large set of correctness properties without the need to generate the network-wide forwarding state from the given configurations.

Batfish [67] encodes routing protocols in Datalog and uses a Datalog solver to check conformance with routing requirements. In Batfish's encoding, the input to the Datalog program is the network configurations and the Datalog solver computes a fixed-point for this program. The fixed-point of the Datalog program is the network-wide forwarding state. Then, Batfish reasons over the fixed-point of the Datalog program to verify various network properties. In `SyNET` we use stratified Datalog to model the network, however, `SyNET` does not use a Datalog solver to compute a fixed-point of the stratified Datalog. Instead, `SyNET` takes an input the fixed-point (i.e., the requirements) and uses SMT solver to synthesize inputs to the stratified Datalog program such that if a Datalog solver runs the Datalog program, then it produces the given requirements.

Minesweeper [73] encodes the network as SMT formulas similar to our work in `NetComplete`. However, the purpose of the two systems is different. Minesweeper can only verify existing configurations, while `NetComplete` can synthesize new configurations and verify existing ones (if the user provides concrete configurations). Both systems build a model of the network $\mathcal{N}$. One of the differences in the encoding is that Minesweeper is more concrete since the configurations are concrete, while `NetComplete`'s model has more free variables that correspond to the holes in the configurations sketch. Given the larger number of free variables, our model can be slower to solve using an SMT solver. Thus, we developed optimization techniques (e.g., CEGIS algorithm and BGP Propagation graphs) to reduce the search time. Without optimizations to reduce the search space, using Minesweeper to synthesize configurations would be very slow for any practical application. Given a Policy φ_R and network model $\mathcal{N}$ (as encoded by Minesweeper or `NetComplete`), we ask the SMT solver slightly different questions. Minesweeper asks if $\mathcal{N} \wedge \neg\varphi_R$ is *unsat* (that the policy is not violated). While `NetComplete` asks $\mathcal{N} \wedge \varphi_R$ is *sat* and read the values assigned free variables of $\mathcal{N}$ (those are the holes in the configuration sketch).

Analysis of Datalog Programs. Datalog has been successfully used to declaratively specify a variety of static analyzers such as points-to analysis and race detection [74, 75] . It has also been used to verify network-wide configurations for protocols such as OSPF and BGP [67]. Recent work [76] has extended Datalog to operate with richer classes of lattice structures. Further, the μZ tool [77] extends the Z3 SMT solver with support for fixed points. The focus of all these works is on computing the fixed point of a program P for a given input I and then checking a property φ on the fixed point. That is, they check whether $[[P]]_I \models \varphi$. All of these works assume that the input is provided a priori. In contrast, SyNET discovers an input that produces a fixed point satisfying a user-provided property.

The algorithm presented in Zhang et al. [75] can be used to check whether certain tuples are not derived for a given set of inputs. Given a Datalog program P (without negation in the literals), a set Q of tuples, and a set $\mathcal{I}$ of inputs, then the algorithm computes the set $Q \setminus \bigcap\{[[P]]_I \mid I \in \mathcal{I}\}$. This algorithm cannot address our problem because it does not support stratified Datalog programs, which are not monotone. While their encoding can be used to synthesize inputs for each stratum of a stratified Datalog program, it supports only negative properties, which require that certain tuples are not derived. Our approach in SyNET is thus more general than [75] and can be used in their application domain.

The FORMULA system [78, 79] can synthesize inputs for non-recursive Datalog programs, as it supports non-recursive Horn clauses with stratified negation (even though [80] which uses FORMULA shows examples of recursive Horn clauses w/o negation). Handling recursion with stratified negation is nontrivial as bounded unrolling is unsound if applied to all strata together. Note that virtually all network specifications require recursive rules, which our system supports.

Symbolic Analysis and Program Synthesis. Our SyNET's synthesis algorithm is similar in spirit to symbolic (or concolic) execution, which is used to generate inputs for programs that violate a given assertion automatically (e.g., division by zero); see [81–83] for an overview. These approaches unroll loops up to a bound and find inputs by calling an SMT solver on the symbolic path. While we also find inputs for a symbolic formula, the entire setting, techniques, and algorithms are all different from the standard symbolic execution setting.

Counter-example guided synthesis approaches are also related [84]. Typically, the goal of program synthesis is to discover a program, while in our case the program is given and we synthesize an input for it. In NetComplete, we showed a novel instantiation of counter-example guided inductive synthesis (CEGIS) [84] for synthesizing weights in OSPF. CEGIS is a general concept that has become popular in the program synthesis community. A key challenge in using it is finding effective ways to specialize it (e.g., efficient representation of the hypothesis space, interaction with the SMT solver) to the particular application domain (e.g., networking and the OSPF protocol in our case).

3 GENERAL NETWORK-WIDE CONFIGURATIONS SYNTHESIS

In this chapter, we present the first general solution to the network-wide configurations synthesis problem. Our solution accepts *(i)* the network specification and *(ii)* the routing requirements as inputs and outputs real network configurations. Considering the network specification as input to our solution allows us to develop a synthesizer that not only works for a limited set of hard-coded protocols but also works for any existing or future protocol(s) specified in the network specification as an input. We remark that the specification is defined once for each protocol; one can imagine a library of protocol specifications available for reuse.

Our approach to solve this problem is based on two steps.

First, we use stratified Datalog to capture the behavior of the network, i.e., the distributed routing protocols ran by the routers in the network and the dependencies between protocols. We refer to this Datalog program as the *network specification* $\mathcal{N}$. Here, the input to the Datalog program represents the network-wide configurations, while the fixed point of a Datalog program represents the stable forwarding state of the network. Datalog is indeed particularly well-suited for describing distributed routing protocols in a declarative way [67, 85].

Second, and a key insight of our work, we pose the network-wide configurations synthesis problem as an instance of finding an input for a stratified Datalog program where the program's fixed point satisfies a given property. In this problem formation, the network operators provide merely the high-level requirements on the forwarding state (i.e., which is the same as requiring the Datalog program's fixed point to satisfy those requirements), and our synthesizer automatically finds an input I to the Datalog program (i.e., which identifies the wanted network-wide configurations). We remark that our Datalog input synthesis algorithm is a general, independent contribution, and is applicable beyond networks.

We start this chapter with a brief introduction to stratified Datalog programs in Section 3.1. Then, in Section 3.2, we show how to express network specification $\mathcal{N}$, the routing requirements φ_R and the network-wide configurations $\mathcal{C}$ declaratively. Afterward, we formulate the network-wide synthesis problem in terms of input synthesis for stratified Datalog in Section 3.3, and we present the first input synthesis algorithm for stratified Datalog in Section 3.4. This algorithm is of broader interest and is applicable beyond networks. In Section 3.5 we describe how to use the Datalog's input synthesis algorithm to synthesis network-wide configurations. In Section 3.6,

$$\begin{array}{rrcl@{\qquad}rrcl}
\textit{(Values)} & v & \in & \textit{Vals} & \textit{(Atom)} & a & ::= & p(\overline{t}) \\
\textit{(Variables)} & X, Y & \in & \textit{Vars} & \textit{(Literal)} & l & ::= & a \mid \neg a \\
\textit{(Term)} & t & ::= & X \mid v & \textit{(Rule)} & r & ::= & a \leftarrow \overline{l} \\
\textit{(Predicates)} & p, q & \in & \textit{Preds} & \textit{(Program)} & P & ::= & \overline{r}
\end{array}$$

FIGURE 3.1: Syntax of stratified Datalog.

we present an instantiation and an end-to-end implementation of our input synthesis algorithm along with network-specific optimizations in a system called `SyNET`. Finally, Section 3.7 concludes this chapter.

3.1 BACKGROUND: STRATIFIED DATALOG

We briefly overview the syntax and semantics of stratified Datalog.

Syntax. Datalog's syntax is given in Figure 3.1. We use $\overline{r}$, $\overline{l}$, and $\overline{t}$ to denote zero or more rules, literals, and terms separated by commas, respectively. A Datalog program is *well-formed* if for any rule $a \leftarrow \overline{l}$, we have $vars(a) \subseteq vars(\overline{l})$, where $vars(\overline{l})$ returns the set of variables in $\overline{l}$.

A predicate is called *extensional* if it appears only in the bodies of rules (right side of the rule), otherwise (if it appears at least once in a rule head) it is called *intensional*. We denote the sets of extensional and intensional predicates of a program P by $edb(P)$ and $idb(P)$, respectively.

A Datalog program P is *stratified* if its rules can be partitioned into strata $P_1, \dots, P_n$ such that if a predicate p occurs in a *positive literal* in the body of a rule in P_i, then all rules with p in their heads are in a stratum P_j with $j \leq i$. While, if a predicate q occurs in a *negative literal* in the body of a rule in P_i, then all rules with q in their heads are in a stratum P_j with $j < i$. Stratification ensures that predicates that appear in negative literals are fully defined in lower strata.

We syntactically extend stratified Datalog with aggregate functions such as `min` and `max`. This extension is possible as stratified Datalog is equally expressive to Datalog with stratified aggregate functions [86].

Semantics. Stratified Datalog's semantics is given in Figure 3.2. Let $\mathcal{A} = \{p(\overline{t}) \mid \overline{t} \subseteq \textit{Vals}\}$ denote the set of all ground (i.e., variable-free) atoms. The complete lattice $(\mathcal{P}(\mathcal{A}), \subseteq, \cap, \cup, \emptyset, \mathcal{A})$ partially orders the set of interpretations $\mathcal{P}(\mathcal{A})$.

Given a substitution $\sigma \in \textit{Vars} \rightarrow \textit{Vals}$ mapping variables to values. Given an atom a, we write $\sigma(a)$ for the ground atom obtained by replacing the variables in a according

$$\begin{array}{rrcl}
\textit{(Substitutions)} & \sigma & \in & \textit{Vars} \to \textit{Vals} \\
\textit{(Ground atoms)} & \mathcal{A} & = & \{p(\bar{t}) \mid \bar{t} \subseteq \textit{Vals}\} \\
\textit{(Consequence} & T_P & \in & \mathcal{P}(\mathcal{A}) \to \mathcal{P}(\mathcal{A}) \\
\textit{operator)} & T_P(A) & = & A \cup \{\sigma(a) \mid a \leftarrow l_1 \dots l_n \in P, \forall l_i \in \bar{l}.\ A \vdash \sigma(l_i)\}, \\
 & & & \textit{where } A \vdash \sigma(a) \text{ if } \sigma(a) \in A \\
 & & & \textit{and } A \vdash \sigma(\neg a) \text{ if } \sigma(a) \notin A \\
\textit{(Input)} & I & \subseteq & \{p(\bar{t}) \mid p(\bar{t}) \in \mathcal{A}, p \in \textit{edb}(P)))\} \\
\textit{(Model)} & [[P]]_I & = & M_n, \\
 & \multicolumn{3}{r}{\textit{where } M_0 = I \text{ and } M_i = \bigcap\{A \in \mathsf{fp}\ T_{P_i} \mid A \subseteq M_{i-1}\}}
\end{array}$$

FIGURE 3.2: Semantics for a Datalog program P with strata $P_1 \cup \dots \cup P_n$.

to σ; e.g., $\sigma(p(X))$ returns the ground atom $p(\sigma(X))$. The consequence operator $T_P \in \mathcal{P}(\mathcal{A}) \to \mathcal{P}(\mathcal{A})$ for a program P:

$$T_P(A) = A \cup \{\sigma(a) \mid a \leftarrow l_1 \dots l_n \in P, \forall l_i \in \bar{l}.\ A \vdash \sigma(l_i)\}$$

where $A \vdash \sigma(a)$ if $\sigma(a) \in A$ and $A \vdash \sigma(\neg a)$ if $\sigma(a) \notin A$.

An input for P is a set of ground atoms constructed using P's extensional predicates. Let P be a program with strata $P_1, \dots, P_n$ and I be an input for P. The model of P for I, denoted by $[[P]]_I$, is M_n, where $M_0 = I$ and $M_i = \bigcap\{A \in \mathsf{fp}\ T_{P_i} \mid A \subseteq M_{i-1}\}$ is the smallest fixed point of T_{P_i} that is greater than the lower stratum's model M_{i-1}.

3.2 DECLARATIVE NETWORK SPECIFICATION

We first describe how to declaratively specify the behavior of the network as a Datalog program. Then, we discuss how to specify routing requirements as constraints over the fixed point of a Datalog program. Finally, we describe how network-wide configurations are expressed as inputs to Datalog programs.

3.2.1 *Specifying Networks*

To capture a network's behavior, we model *(i)* the behavior of the routing protocols and their interactions, and *(ii)* the topology of the network.

Expressing Network Specification in Stratified Datalog. We formalize the network specification $\mathcal{N}$ as a stratified Datalog program P. The Datalog program P derives the

```
1 OSPFRoute(TC, Router, NextHop, Cost) ←
        SetNetwork(Router, TC),
        SetOSPFEdgeCost(Router, NextHop=⊥, Cost=0)
2 OSPFRoute(TC, Router, NextHop, Cost) ←
        Cost=Cost1+Cost2,
        SetOSPFEdgeCost(Router, NextHop, Cost1),
        OSPFRoute(TC, NextHop, R', Cost2)
3 minOSPF(TC, Router, min<Cost>) ←
        OSPFRoute(TC, Router, NextHop, Cost)
4 BestOSPFRoute(TC, Router, NextHop) ←
        minOSPF(TC, Router, Cost),
        OSPFRoute(TC, Router, NextHop, Cost)
5 Route(TC, Router, Next, "static") ←
        SetStatic(TC, Router, NextHop)
6 Route(TC, Router, NextHop, "ospf") ←
        BestOSPFRoute(TC, Router,  NextHop)
7 minAD(TC, Router, min<Cost>) ←
        Route(TC, Router, NextHop, Proto),
        SetAD(Proto, Router, Cost)
8 Fwd(TC, Router, NextHop) ←
        Route(TC, Router, NextHop, Proto),
        SetAD(Proto, Router, Cost),
        minAD(TC, Router, Cost)
```

FIGURE 3.3: Partial network specification $\mathcal{N}$ expressed as Datalog program. This program specifics OSPF, static routes, and selecting the best route to be the forwarding entry based on the Administrative Distance (AD).

predicate `Fwd(TC, Router, NextHop)` for each traffic class `TC` at each router `Router`. The union of all `Fwd` predicates represents the network's global forwarding state.

As an example, we show (a subset of) of our OSPF formalization in stratified Datalog in Figure 3.3. Furthermore, we show the relevant rules that define how the routing entries computed by OSPF are combined with those defined via static routes to derive the `Fwd` predicates. We do not show the formalization of BGP for brevity.

In the OSPF model, specified in Figure 3.3, there are two extensional (*edb*) predicates `SetOSPFEdgeCost` and `SetNetwork` that are given by the network configurations as input (or to be synthesized by our system). In more details, the *edb* predicate `SetOSPFEdgeCost(R1, R2, Cost)` represents that the routers `R1` and `R2` are neighbors connected by a link with cost `Cost`, while, the *edb* predicate `SetNetwork(Router, TC)` represents that `Router` is directly connected (i.e., advertise) traffic class `TC`. In our example, a traffic class is a set of one or more prefixes.

The first four lines, in Figure 3.3, concisely implement the shortest-path computation performed by the routers running OSPF. More specifically, the first rule drives the predicate `OSPFRoute` for directly connected traffic classes `TC`; i.e., OSPF can forward traffic to prefixes directly attached to a router with zero cost. The second rule transitively computes multi-hop routing paths by summing up the costs associated along all OSPF routes. These two rules combined derive the values for the intensional predict `OSPFRoute`. The third rule, `minOSPF`, derives the minimum cost OSPF route for each router and each destination network by aggregating over all possible OSPF routes. The predicate `BestOSPFRoute(TC, Router, NextHop)` represents the best OSPF route selected by the router `Router` for the traffic class `TC` to be the next hop `NextHop`.

At lines 5 and 6 in Figure 3.3, the predicate `Route(TC, Router, NextHop, Proto)` captures the routing table. In our example, it captures static routes and the best OSPF routes (we do not show BGP). It derives the next hops `Next` for a given traffic class `TC` on a given router `Router` and defines through which protocol `Proto` this route has been learned.

Finally, lines 7 and 8 captures the forwarding state of the network. The Datalog rule in line 7 states that routers select, for each traffic class `TC`, the forwarding entry with the minimal administrative cost `minAD` calculated over all protocols via the Datalog rule in line 8.

For more details on specifying routing protocols in Datalog, see works related to declarative networking [85] and Datalog-based network verification [67].

Network Topology. The network topology T is also captured via Datalog rules (we do not show this in Figure 3.3 for brevity). We model each router as a constant and use predicates to represent the topology. For example, the predicate `SetLink(R1, R2)` represents that two routers $R1$ and $R2$ are connected via a link, and we add the Datalog rule `SetLink(R1, R2)` $\leftarrow$ `true` to define such a link.

3.2.2 *Specifying Requirements*

We specify the requirements as function-free first-order constraints over the predicate `Fwd(TC, Router, NextHop)`, which defines the network's forwarding state. We write $A \vDash \varphi$ to denote that a Datalog interpretation A (i.e., inputs) satisfies φ. For illustration, we describe how common routing requirements can be specified:

`Path(TC, R1, [R1, R2, .., Rn])`: The *path requirement* requirement stipulates that packets for traffic class `TC` must follow the path `R1, .., Rn`. These requirements are specified as a conjunction over the predicate `Fwd`; i.e., `Fwd(TC, R1, R2)` $\wedge \cdots \wedge$ `Fwd(TC, Rn-1, Rn)`.

$\forall$`R1,R2. Fwd(TC1, R1, R2)` $\Rightarrow \neg$`Fwd(TC2, R1, R2)`: The *traffic isolation* requirement stipulates that the paths for two distinct traffic classes `TC1` and `TC2` do not share links in the same direction.

`Reach(TC, R1, R2)`: The *reachability* requirements stipulates that packets for traffic class `TC` can reach router `R2` from router `R1`. The predicate `Reach` is the transitive closure over the predicate `Fwd` that is defined via Datalog rules.

$\forall$`TC,R.` $(\neg$`Reach(TC, R, R)`$)$: The *loop-freeness* is a generic requirement stipulating that the forwarding plane has no loops. This requirement can be specified as follow: $\forall$`TC,R.` $(\neg$`Reach(TC, R, R)`$)$.

More complex requirements, such as way pointing, can be specified based on the core function-free first-order constraints provided by `SyNET`. Further, `SyNET` can be used as a backend for a high-level requirements language that is easier to use by a network operator.

3.2.3 *Network-wide Configurations*

We represent the input (configurations) to the protocols deployed in the network as *edb* predicates. Note, the Datalog program P that formalizes the protocols takes the *edb* predicates as input. For instance, the *edb* predicate `SetOSPFEdgeCost` specifies the OSPF weights associated with the links connected to router in the network. Further, a static route in the router's configuration is represented with the *edb* predicate `SetStatic(TC, Router, NextHop)`. We refer to all local router configurations as the *network-wide configurations* $\mathcal{C}$.

We remark that network-wide configurations must satisfy well-formedness constraints φ_C, which can be directly specified as constraints over the *edb* predicates that represent configurations. For example, the costs in OSPF configurations must be positive integers with a value less than 2^{16}. This is formalized as:

$$\begin{array}{rll} & \forall \texttt{R1,R2,Cost.} & (\texttt{SetOSPFEdgeCost(R1, R2, Cost)} \Rightarrow \texttt{Cost} > 0) \\ \wedge & \forall \texttt{R1,R2,Cost.} & (\texttt{SetOSPFEdgeCost(R1, R2, Cost)} \Rightarrow \texttt{Cost} < 2^{16}) \end{array}$$

To show how the network specification $\mathcal{N}$ accepts the configurations as an input, let us consider the $\mathcal{C}_{new}$ in Figure 2.4 as example. We show the encoding of $\mathcal{C}_{new}$ as *edb* predicts in Figure 3.4. Here, $\mathcal{N}$ takes the predicate `SetStatic(N2, A, D)`, which represents static routes, defines a static route for `N2` from A to D. Additionally, `SetAD` defines the administrative cost of static routes to be lower than that of OSPF (so static routes are preferred over forwarding entries computed by OSPF). Moreover, the predict `SetOSPFEdgeCost` defines the link cost for every internal link the topology. Similarly, BGP configurations, not shown in the example, can be specified as *edb* predicts. Note, in synthesis $\mathcal{C}$ is not known.

```
SetNetwork(D, N1)
SetNetwork(D, N2)
SetStatic(N2,A,D)
SetAD(A,5,"static")
SetAD(A,10,"ospf")
SetOSPFEdgeCost(A,B,9)
SetOSPFEdgeCost(A,C,10)
SetOSPFEdgeCost(A,D,20)
SetOSPFEdgeCost(B,C,5)
SetOSPFEdgeCost(B,D,7)
SetOSPFEdgeCost(C,D,5)
```

FIGURE 3.4: Example configurations input to the network specification $\mathcal{N}$ defined in Figure 3.3 based on $\mathcal{C}_{new}$ defined in Figure 2.4.

The goal of a network-wide configurations synthesis framework to find *edb* predicates that then transformed (via simple rewrite rules) to vendor-specific routers configurations.

3.3 REDUCING NETWORK-WIDE CONFIGURATIONS SYNTHESIS PROBLEM TO STRATIFIED DATALOG INPUT SYNTHESIS PROBLEM

In this section, we pose the network-wide synthesis problem as the problem of input synthesis for stratified Datalog.

Formally, the network-wide configurations synthesis problem that we consider in this chapter is as follow. Given the two inputs a declarative network specification $\mathcal{N}$ and a high-level routing requirements φ_R, find a complaint configurations $\mathcal{C}$ such that using the new configurations, the distributed routing protocols converge to a network-wide forwarding state that satisfies the input requirements. We write $[[\mathcal{N}]]_{\mathcal{C}} \vDash \varphi_R$ to denote that a network specified by $\mathcal{N}$ converges to a forwarding state complaint with φ_R given a configuration $\mathcal{C}$, for short we write this as $\mathcal{C} \vDash \varphi_R$.

To reduce the network-wide configurations synthesis $[[\mathcal{N}]]_{\mathcal{C}} \vDash \varphi_R$ to input synthesis for stratified Datalog, we express the network specification $\mathcal{N}$ as a stratified Datalog program P. The semantics of P is given by a fixed point, which is computed by first partitioning P's rules into strata $P_1, \ldots, P_n$ and the iteratively computing, for each startum P_i, the least fixed point that contains the least fixed point of the previous stratum. We denote the resulting fixed point by $[[P]]$.

An input I for P is a set of tuples constructed using P's *edb* predicate symbols; i.e., those that do not appear in the heads of P's rules. We denote the fixed point obtained when evaluating P using I by $[[P]]_I$. In Section 3.2.3, we present how a network configurations C is expressed as *edb* predicates. Likewise, using simple transformations, the synthesized Datalog input I can be directly used to derive vendor-specific router configurations.

A property φ specifies both which tuples *must* be contained in P's fixed point as well as which tuples *must not* be contained in the fixed point. Given a property φ, a stratified Datalog program P, and an input I for P, we write $[[P]]_I \models \varphi$ to denote that $[[P]]_I$ satisfies φ. In Section 3.2.2, we show how to derive φ from φ_R by expressing φ_R as function-free first-order constraints over the predicate `Fwd`.

Definition 3.3.1. *Input Synthesis for Stratified Datalog*
Input: A stratified Datalog program P and a property φ.
Output: An input I such that $[[P]]_I \models \varphi$ or *unsat* if no such input exists.

Note that while some Datalog inputs produce a fixed point compatible with the requirements, they are not valid network configurations. For instance, a synthesized input that contains negative OSPF link weights cannot produce a valid network configurations. A synthesized input is well-formed if it derives valid network configurations. In our synthesizer, we add additional constraints φ_C to ensure that I is always well-formed ($I \models \varphi_C$). For our synthesis problem we use: $\varphi = \varphi_R \wedge \varphi_C \wedge T$. These conditions guarantee that: *(i)* the synthesized configuration is well-formed ($I \models \varphi_C$), *(ii)* they are compatible with the network topology ($I \models T$), and *(iii)* the network's forwarding state satisfies the requirements ($[[\mathcal{N}]]_C \models \varphi_R$).

We remark on several key points. First, for any *edb* atom a, we have $a \in I$ *iff* $a \in [[P]]_I$. Therefore, $[[P]]_I \models T \wedge \varphi_C$ implies $I \models T \wedge \varphi_C$, and it is thus equivalent to find an input I such that $[[P]]_I \models \varphi_R \wedge T \wedge \varphi_C$. Second, the conjunction $\varphi_R \wedge T \wedge \varphi_C$ can be encoded through Datalog rules, such that an atom $a_{SAT} \in [[\mathcal{N}]]_I$ *iff* $[[\mathcal{N}]]_I \models \varphi_R \wedge T \wedge \varphi_C$. We remark that the satisfiability of network configuration synthesis can thus be reduced to query satisfiability in stratified Datalog [87], and vice versa, which means that our synthesis problem is undecidable. The problem is, however, decidable if we fix a finite set of values to bound the set of inputs. This is a reasonable assumption in the context of networks where values represent finitely many routers and configuration parameters.

3.4 INPUT SYNTHESIS FOR STRATIFIED DATALOG

We present a new iterative algorithm for synthesizing inputs for stratified Datalog. We first describe the high-level flow of the algorithm before presenting the details.

High-Level Flow. Consider the stratified Datalog program P with strata P_1, P_2, and P_3, depicted in Figure 3.5. Incoming and outgoing edges of a stratum P_i indicate the *edb* predicates and, respectively, the *idb* predicates of that stratum. For example, the

$$\begin{tikzcd}
I \arrow[r, "p(\bar{t})"'] \arrow[rr, bend left, "p(\bar{t})"] & \boxed{P_1} \arrow[r, "q(\bar{t})"'] \arrow[rr, bend left, "q(\bar{t})"] & \boxed{P_2} \arrow[r, "r(\bar{t})"'] & \boxed{P_3} \arrow[r, "s(\bar{t})"] & {}
\end{tikzcd}$$

FIGURE 3.5: A Datalog program P with strata P_1, P_2, and P_3, and the flow of predicates between the strata.

stratum P_3 takes as input predicates $q(\bar{t})$ and $r(\bar{t})$ and derives the predicate $s(\bar{t})$. Our iterative algorithm first synthesizes an input I_3 for P_3 which determines the predicates $q(\bar{t})$ and $r(\bar{t})$ that $P_1 \cup P_2$ must output. To synthesize such an input for a single stratum, we present an algorithm, called $\mathcal{S}_{SemiPos}$, that addresses the input synthesis problem for semi-positive Datalog programs [88]; i.e., Datalog programs where negation is restricted to *edb* predicates. Note that each stratum of a stratified Datalog program is a semi-positive Datalog program. After synthesizing an input I_3 for P_3, our iterative algorithm synthesizes an input I_2 for P_2 such that the fixed-point $[[P_2]]_{I_2}$ produces the predicates $r(\bar{t})$ that are contained in the already synthesized input I_3 for P_3. We note that this iterative process may require backtracking, in case no input for P_2 can produce the desired predicates $r(\bar{t})$ contained in I_3. The algorithm terminates when it synthesizes inputs for all three strata.

Next, we first present the algorithm $\mathcal{S}_{SemiPos}$ that is used to synthesize an input for a single stratum (a semi-positive Datalog program). Then, we present the general algorithm, called $\mathcal{S}_{Strat}$, that iteratively applies $\mathcal{S}_{SemiPos}$ for each stratum to synthesize inputs for stratified Datalog programs.

3.4.1 *Input Synthesis for Semi-positive Datalog with SMT*

The key idea of our approach is to reduce the input synthesis problem to satisfiability of SMT constraints: Given a semi-positive Datalog program P and a constraint φ, we encode the question $\exists I.\ [[P]]_I \vDash \varphi$ into an SMT constraint ψ. If ψ is satisfiable, then from a model of ψ we can derive an input I such that $[[P]]_I \vDash \varphi$.

SMT Encoding Challenges. Given a Datalog program P and a constraint φ, encoding the question $\exists I.\ [[P]]_I \vDash \varphi$ with SMT constraints is non-trivial due to the mismatch between Datalog's program fixed point semantics and the classical semantics of first-order logic. This means that simply taking the conjunction of all Datalog rules into an

SMT solver does not solve our problem. For example, consider the following Datalog program P_{tc}:

$$\begin{array}{rcl} tc(X,Y) & \leftarrow & e(X,Y) \\ tc(X,Y) & \leftarrow & tc(X,Z), tc(Z,Y) \end{array}$$

which computes the transitive closure of the predicate $e(X,Y)$. A naive way of encoding these Datalog rules with SMT constraints:

$$\begin{array}{rcl} \forall X,Y.\,(e(X,Y) & \Rightarrow & tc(X,Y)) \\ \forall X,Y.\,((\exists Z.\, tc(X,Z) \wedge tc(Z,Y)) & \Rightarrow & tc(X,Y)) \end{array}$$

and we denote the conjunction of these two SMT constraints as $[P_{tc}]$. Now, suppose we have the fixed point constraint $\varphi_{tc} = (\neg e(v_0,v_2)) \wedge tc(v_0,v_2)$ and we want to generate an input I so that $[\![P_{tc}]\!]_I \models \varphi_{tc}$. A model that satisfies $[P_{tc}] \wedge \varphi_{tc}$ is

$$\mathcal{M} = \{e(v_0,v_1), tc(v_0,v_1), tc(v_0,v_2)\}$$

The input derived from this model, obtained by projecting $\mathcal{M}$ over the *edb* predicate e, is $I_{\mathcal{M}} = \{e(v_0,v_1)\}$. If we run P_{tc} with inputs $I_{\mathcal{M}}$, we get

$$[\![P_{tc}]\!]_{I_{\mathcal{M}}} = \{e(v_0,v_1), tc(v_0,v_1)\}$$

and so $[\![P_{tc}]\!]_{I_{\mathcal{M}}} \not\models \varphi_{tc}$, which is clearly not what is intended.

SMT Encoding. Our key insight for encoding is to split the constraint φ into a conjunction of positive and negative clauses, where a clause φ is positive (respectively, negative) if $A \models \varphi$ implies that $A' \models \varphi$ for any interpretation $A' \supseteq A$ (respectively, $A' \subseteq A$). We can then unroll recursive predicates to obtain a sound encoding for positive constraints, and we do not unroll them to get a sound encoding for negative constraints.

The encoding of a Datalog program P into an SMT constraint is defined in Figure 3.6. The resulting SMT constraint is denoted by $[P]_k$, where the parameter k defines the number of unroll steps. In the encoding we assume that *(i)* all terms in rules' heads are variables and *(ii)* rules' heads with the same predicate have identical variable names. Note, Datalog programs can be converted into this form using rectification [89] and variable renaming.

Function Encode. The constraint returned by $\textsc{Encode}(p,P)$ states that an atom $p(X)$ is derived if P has a rule that derives $p(\overline{X})$ and whose body evaluates to true. To capture Datalog's semantics, the variables, $\overline{X}$, in $p(\overline{X})$ are universally quantified, while those in the rules' bodie, $\overline{Y} = vars(\overline{l}) \setminus \overline{X}$, are existentially quantified. This constraint $\textsc{Encode}(p,P)$ is sound for negative requirements, but not for positive

$$
\begin{array}{lcl}
[P]_k & = & \bigwedge\limits_{p \in idb(P)} \textsc{Encode}(P,p) \wedge \textsc{Unroll}(P,p,k) \\
\textsc{Encode}(P,p) & = & \bigwedge\limits_{p(\overline{X}) \leftarrow \overline{l} \in P} \forall \overline{X}.\ \big((\exists \overline{Y}.\ \bigwedge \overline{l}) \Rightarrow p(\overline{X})\big), \\
 & & \textit{where } \overline{Y} = vars(\overline{l}) \setminus \overline{X} \\
\textsc{Unroll}(P,p,k) & = & \bigwedge\limits_{0<i\leq k} \textsc{Step}(P,p,i) \\
\textsc{Step}(P,p,i) & = & \forall \overline{X}.\ \big(p_i(\overline{X}) \Leftrightarrow (\bigvee\limits_{p(\overline{X}) \leftarrow \overline{l} \in P} \exists \overline{Y}.\ \tau(\overline{l}, i-1))\big), \\
 & & \textit{where } \overline{Y} = vars(\overline{l}) \setminus \overline{X} \\
\tau(\overline{l},k) & = & \begin{cases} \tau(l_1,k) \wedge \cdots \wedge \tau(l_n,k) & \text{if } \overline{l} = l_1 \wedge \cdots \wedge l_n \\ \neg\tau(p(\overline{t}),k) & \text{if } \overline{l} = \neg p(\overline{t}) \\ \mathsf{false} & \text{if } \overline{l} = p(\overline{t}), p \in idb(P), k = 0 \\ p_k(\overline{t}) & \text{if } \overline{l} = p(\overline{t}), p \in idb(P), k > 0 \\ p(\overline{t}) & \text{if } \overline{l} = p(\overline{t}), p \in edb(p) \end{cases}
\end{array}
$$

FIGURE 3.6: Encoding a Datalog program P into SMT constraints $[P]_k$.

ones as it does not state that $p(\overline{X})$ is derived *only if* a rule body with $p(\overline{X})$ in the head evaluates to true.

Functions Unroll and Step. The SMT constraint returned by $\textsc{Step}(P,p,i)$ encodes whether an atom $p(X)$ is derived after i applications of P's rules; e.g., $p(X)$'s truth value after 3 steps is represented with the atom $p_3(X)$. Intuitively, $p(X)$ is true *iff* there is a rule that derives $p(X)$ and whose body evaluates to true using the atoms derived in previous iterations. Which atoms are derived in previous iterations is captured by the literal renaming function τ. Note that $\tau(l,0)$ returns false for any *idb* literal l since all intensional predicates are initially false. Further, $\tau(l,k)$ returns l for any extensional literal l (the last case in Figure 3.6) since their truth values do not change. Finally, the constraint returned by $\textsc{Unroll}(P,p,k)$ conjoins $\textsc{Step}(P,p,0)$, …, $\textsc{Step}(P,p,k)$ to capture the derivation of $p(X)$ after k steps. This is sound for positive requirements, but not for negative ones since more $p(X)$ atoms may be derived after k steps.

Example. To illustrate the encoding, we translate the Datalog program:

$$
\begin{array}{lcl}
tc(X,Y) & \leftarrow & e(X,Y) \\
tc(X,Y) & \leftarrow & tc(X,Z), tc(Z,Y)
\end{array}
$$

Algorithm 1: Algorithm $\mathcal{S}_{SemiPos}$ for semi-positive Datalog

```
Input: Semi-positive Datalog program $P$ and a constraint $\varphi$
Output: An input $I$ such that $[[P]]_I \models \varphi$ or $\bot$
1 begin
2     $\varphi' \leftarrow$ Simplify($\varphi$)
3     for $k \in [1..bound_k]$ do
4         $\varphi_k \leftarrow$ Rewrite($\varphi', k$)
5         $\psi \leftarrow [P]_k \wedge \varphi_k$
6         if $\exists J.\ J \models \psi$ then
7             $I \leftarrow \{p(\bar{t}) \in J \mid p \in edb(P)\}$
8             return $I$
9     return $\bot$
```

which computes the transitive closure of the predicate $e(X,Y)$. This program has one *idb* predicate, tc. The function Encode(P, tc) returns

$$\begin{array}{rcl} (\forall X,Y.\ e(X,Y) & \Rightarrow & tc(X,Y)) \\ \wedge(\forall X,Y.\ (\exists Z.\ tc(X,Z) \wedge tc(Z,Y)) & \Rightarrow & tc(X,Y)) \end{array}$$

We apply function Unroll$(P, tc, 2)$ for $k = 2$, which after simplifications returns

$$\begin{array}{lcl} \forall X,Y.\ (tc_1(X,Y) & \Leftrightarrow & e(X,Y)) \\ \forall X,Y.\ (tc_2(X,Y) & \Leftrightarrow & e(X,Y) \vee (\exists Z.\ tc_1(X,Z) \wedge tc_1(Z,Y)) \end{array}$$

In the constraints, the predicates tc_1 and tc_2 encode the derived predicates tc after 1 and, respectively, 2, derivation steps.

Algorithm. $\mathcal{S}_{SemiPos}(P,\varphi)$, see Algorithm 1, first calls function Simplify(φ) that *(i)* instantiates any quantifiers in φ and *(ii)* transforms the result into a conjunction of clauses, where each clause is a disjunction of literals.

Afterward, the algorithm iteratively unrolls the Datalog rules, up to a pre-defined bound $bound_k$. In each step of the for-loop, the algorithm $\mathcal{S}_{SemiPos}(P,\varphi)$ generates an SMT constraint that captures *(i)* which atoms are derived after k applications of P's rules and *(ii)* which atoms are never derived by P. The resulting SMT constraint is denoted by $[P]_k$. The algorithm also rewrites the simplified constraint φ' using the function Rewrite(φ', k) which recursively traverses conjunctions and disjunctions in the simplified constraint φ' and maps positive literals to the k-unrolled predicate $p_k(\bar{t})$ and negative literals to $\neg p(\bar{t})$:

$$\textsc{Rewrite}(\varphi, k) = \begin{cases} p_k(\bar{t}) & \text{if } \varphi = p(\bar{t}) \\ \neg p(\bar{t}) & \text{if } \varphi = \neg p(\bar{t}) \\ \textsc{Rewrite}(\varphi_1, k) \vee \cdots \vee \textsc{Rewrite}(\varphi_n, k) & \text{if } \varphi = \varphi_1 \vee .. \vee \varphi_n \\ \textsc{Rewrite}(\varphi_1, k) \wedge \cdots \wedge \textsc{Rewrite}(\varphi_n, k) & \text{if } \varphi = \varphi_1 \wedge .. \wedge \varphi_n \end{cases}$$

Note that since $\vee$ and $\wedge$ are monotone, negative literals constitute negative constraints and positive literals constitute positive constraints.

If the resulting constraint $[P]_k \wedge \psi_k$ is satisfiable, then an input is derived by projecting the interpretation I that satisfies the constraint over all *edb* predicates. Note that if there is an input I such that $[[P]]_I \vDash \varphi$ and for which the fixed point $[[P]]_I$ is reached in less than $bound_k$ steps, then $\mathcal{S}_{SemiPos}(P, \varphi)$ is guaranteed to return an input.

Theorem 3.4.1. Let P be a semi-positive Datalog program, φ a constraint. If $\mathcal{S}_{SemiPos}(P, \varphi) = I$ then $[[P]]_I \vDash \varphi$.

Next, we prove the correctness of the semi-positive algorithm.

Semi-positive Algorithm Proof

Given a program P and an interpretation J, we denote by $inp(P, J)$ the set of all ground atoms contained in J that are constructed with *edb* predicate symbols of the program P.

First, we remark that any semi-positive Datalog program can be stratified into a single partition P. The model $[[P]]_I$ of P for a given input I is given by the least fixed point of the consequence operator T_P that contains I. The fixed point $[[P]]_I$ can be iteratively computed as $T^{\infty}_{P,I}$ where $T^0_{P,I} = I$ and $T^{i+1}_{P,I} = T_P(T^i_{P,I}) \cup T^i_{P,I}$. Note that $T^i_{P,I} \subseteq T^j_{P,I}$ for any $i \leq j$.

Negative Constraints. We first show that any interpretation J that satisfies the constraint $[P]_k$ is an over-approximation of the ground atoms $p(\bar{t})$ derived by program P for the input $inp(P, J)$.

Lemma 3.4.2. Let P be a semi-positive Datalog program. For any $k \geq 0$ and any interpretation J such that $J \vDash [P]_k$, we have $[[P]]_{inp(P,J)} \subseteq J$.

Proof. By induction on the iterative computation of $[[P]]_{inp(P,J)}$, we show that for any $i \geq 0$ we have $T^i_{P,inp(P,J)} \subseteq J$.

Base Case: For the base case, we have $i = 0$. Then, $T^0_{P,inp(P,J)} = inp(P,J)$. Since $inp(P,J) = \{p(\bar{t}) \in J \mid p \in edb(P)\}$, it is immediate that $inp(P,J) \subseteq J$, and thus $T^0_{P,inp(P,J)} \subseteq J$.

Inductive Step: For our inductive step, assume that $T^j_{P,inp(P,J)} \subseteq J$ holds for $0 \leq j \leq i$, for some $i \geq 0$. We show that $T^{i+1}_{P,inp(P,J)} \subseteq J$.

By definition, we have $T^{i+1}_{P,inp(P,J)} = T_P(T^i_{P,inp(P,J)}) \cup T^i_{P,inp(P,J)}$. By induction, we know that $T^j_{P,inp(P,J)} \subseteq J$. It remains to prove that $T_P(T^i_{P,inp(P,J)}) \subseteq J$. Suppose $p(\bar{t}) \in T_P(T^i_{P,inp(P,J)})$. We need to show that $p(\bar{t}) \in J$. Since $p(\bar{t}) \in T_P(T^i_{P,inp(P,J)})$, we know that there is a rule $p(\overline{X}) \leftarrow l_1, \ldots, l_n$ in P such that for some substitution σ we have $\sigma(p(\overline{X})) = p(\bar{t})$ and for all l_i we have $T^i_{P,inp(P,J)} \vdash \sigma(l_i)$. We can conclude that $T^i_{P,inp(P,J)} \models \exists\overline{Y}.l_1 \wedge \cdots \wedge l_n$. By induction hypothesis, we have $T^j_{P,inp(P,J)} \subseteq J$. Since P is semi-positive, we know that all negative literals in $l_1, \ldots, l_n$ are constructed using *edb* predicates. Moreover, both $T_{P,inp(P,J)}$ and J contain the same set of *edb* literals, and we can thus conclude that $J \models \exists\overline{Y}.l_1 \wedge \cdots \wedge l_n$. By definition of $[P]_k$, we know that $[P]_k$ contains the constraint $\forall\overline{X}. ((\exists\overline{Y}.l_1 \wedge \cdots \wedge l_n) \Rightarrow p(\overline{X}))$. Since $J \models [P]_k$, we get that $J \models p(\bar{t})$. Therefore, $p(\bar{t}) \in J$. □

We can now prove that $\mathcal{S}_{SemiPos}$ is sound for negative constraints.

Lemma 3.4.3. Let P be a semi-positive Datalog program and $\neg p(\bar{t})$ a negative constraint. If $\mathcal{S}_{SemiPos}(P, \neg p(\bar{t})) = I$, then $[\![P]\!]_I \models \neg p(\bar{t})$.

Proof. Suppose $\mathcal{S}_{SemiPos}$ returns an input I for some $k \in [1..bound_k]$. The input I is derived from an interpretation J such that $J \models [P]_k \wedge \neg p(\bar{t})$ and $inp(P,J) = I$. From $J \models \neg p(\bar{t})$, we get $p(\bar{t}) \notin J$. Furthermore, from $J \models [P]_k$, by Lemma 3.4.2, we get $[\![P]\!]_I \subseteq J$. We conclude that $p(\bar{t}) \notin [\![P]\!]_I$ and thus $[\![P]\!]_I \models \neg p(\bar{t})$. □

Positive Constraints. We now prove that any interpretation J that satisfies the constraint $[P]_k$ contains a ground atom $p_k(\bar{t})$ then the ground atom $p(\bar{t})$ is derived by P for input $inp(P,J)$.

Lemma 3.4.4. Let P be a semi-positive Datalog program. For any $k \geq 1$ and any interpretation J such that $J \models [P]_k$, if $p_k(\bar{t}) \in J$ then $p(\bar{t}) \in [\![P]\!]_{inp(P,J)}$.

Proof. By induction on the iterative computation of $[\![P]\!]_I$, we show that $p_i(\bar{t}) \in J$ implies that $p(\bar{t}) \in T^i_{P,inp(P,J)}$, for any $i \geq 1$. Since $T^i_{P,inp(P,J)} \subseteq [\![P]\!]_{inp(P,J)}$ for any i, this also implies that $p(\bar{t}) \in [\![P]\!]_{inp(P,J)}$.

Base Case: For the base case, we have $i = 1$. Assume $p_1(\bar{t}) \in J$. By definition of $[P]_1$, the constraint

$$\forall \overline{X}.\big(p_1(\overline{X}) \Leftrightarrow (\bigvee_{p(\overline{X}) \leftarrow \bar{l} \in P} \exists \overline{Y}.\, \tau(\bar{l}, 0))\big),$$

where $\overline{Y} = vars(\bar{l}) \setminus \overline{X}$, is conjoined to the constraint $[P]_1$. Since $J \models p_1(\bar{t})$, we conclude that there is a rule $p(\overline{X}) \leftarrow l_1, \ldots, l_n$ in P such that for some substitution σ we have $\sigma(p(\overline{X})) = p(\bar{t})$ and $J \models \sigma(\tau(l_i, 0))$ for $1 \le i \le n$. By definition of τ, all literals l_i must be constructed using *edb* predicates (since $\tau(l_i, 0)$ maps any idb literal l_i to false and $J' \not\models$ false for any J'). Note that for *edb* literals we have $\tau(l_i, 0) = l_i$. Since J and $inp(P, J)$ contain the same set of *edb* ground atoms, we get $inp(P, J) \vdash \sigma(l_i)$ for all $1 \le i \le n$. By definition of T_P and $T^1_{P,inp(P,J)}$, it is immediate that $p(\bar{t}) \in T^1_{P,inp(P,J)}$.

Inductive Step: For our inductive step, assume that $p_j(\bar{t}) \in J$ implies that $p(\bar{t}) \in T^j_{P,inp(P,J)}$, for $1 \le j \le i$, for some $i \ge 1$. We show that $p_{i+1}(\bar{t}) \in J$ implies that $p(\bar{t}) \in T^{i+1}_{P,inp(P,J)}$.

Assume $p_{i+1}(\bar{t}) \in J$. By definition of $[P]_{i+1}$, the constraint

$$\forall \overline{X}.\big(p_{i+1}(\overline{X}) \Leftrightarrow (\bigvee_{p(\overline{X}) \leftarrow \bar{l} \in P} \exists \overline{Y}.\tau(\bar{l}, i))\big),$$

where $\overline{Y} = vars(\bar{l}) \setminus \overline{X}$, is conjoined to the constraint $[P]_{i+1}$. Since $p_{i+1}(\bar{t}) \in J$ and $J \models [P]_{i+1}$, we know there is a rule $p(\overline{X}) \leftarrow l_1, \ldots, l_n$ in P such that for some substitution σ we have $\sigma(p(\overline{X})) = p(\bar{t})$ and $J \models \sigma(\tau(l_1, i)) \wedge \cdots \wedge \sigma(\tau(l_n, i))$. For any *edb* literal l in the body of this rule, we have $\tau(l, i) = l$ and $\sigma(l) \in J$ iff $\sigma(l) \in T^i_{P,inp(P,J)}$, simply because J and $T^i_{P,inp(P,J)}$ contain the same *edb* ground atoms. Furthermore, for any positive *idb* literal $\sigma(l) = q(\overline{t'}) \in J$ in the body of this rule, we have $\tau(q(\overline{t'}), i) = q_i(\overline{t'})$ and using our inductive hypothesis we get $q(\bar{t}) \in T^i_{P,inp(P,J)}$. We conclude for all literals l that appear in the body of this rule we have $T^i_{P,inp(P,J)} \vdash \sigma(l)$. By definition of $T^{i+1}_{P,inp(P,J)}$ and T_P we conclude that $p(\bar{t}) \in T^{i+1}_{P,inp(P,J)}$. □

We can now prove that $\mathcal{S}_{SemiPos}$ is sound for positive constraints.

Lemma 3.4.5. Let P be a semi-positive Datalog program and $p(\bar{t})$ a positive constraint. If $\mathcal{S}_{SemiPos}(P, p(\bar{t})) = I$ then $[\![P]\!]_I \models p(\bar{t})$.

Proof. Suppose $\mathcal{S}_{SemiPos}$ returns an input I for some $k \in [1..bound_k]$. The input I is derived from an interpretation J such that $inp(P, J) = I$ and $J \models [P]_k \wedge p_k(\bar{t})$.

From $J \vDash p_k(\bar{t})$, we know that $p_k(\bar{t}) \in J$. From $J \vDash [P]_k$, by Lemma 3.4.4, we get $p(\bar{t}) \in [\![P]\!]_{inp(P,J)}$. It is immediate that $[\![P]\!]_I \vDash p(\bar{t})$. □

We can now prove the correctness of $\mathcal{S}_{SemiPos}$ Theorem 3.4.1.

Proof. The algorithm $\mathcal{S}_{SemiPos}$ transforms the constraint φ into a constraint that uses conjunction and disjunction over positive and negative constraints. Since conjunction and disjunction and monotone, the proof of $[\![P]\!]_I \vDash \varphi$ follows from Lemma 3.4.3 and Lemma 3.4.5. □

3.4.2 *Iterative Input Synthesis for Stratified Datalog*

Our iterative input synthesis algorithm for stratified Datalog $\mathcal{S}_{Strat}$ is given in Algorithm 2. We assume that the fixed point constraint φ is defined over predicates that appear in the highest stratum P_n; this is without any loss of generality, as any constraint can be expressed using Datalog rules in the highest stratum, using a standard reduction to query satisfiability; see [87]. Starting with the highest stratum P_n, $\mathcal{S}_{Strat}$ generates an input I_n for P_n such that $[\![P_n]\!]_{I_n} \vDash \varphi$. Then, it iteratively synthesizes an input for the lower strata $P_{n-1}, \dots, P_1$ using the algorithm $\mathcal{S}_{SemiPos}$. Finally, to construct an input for P, the algorithm combines the inputs synthesized for all strata and returns this.

Recall that the fixed point of a stratum P_i is given as input to the higher strata P_{i+1}, $\dots, P_n$. A key step when synthesizing an input I_i for P_i is to ensure that the *idb* predicates derived by P_i are identical to the *edb* predicates synthesized for the inputs $I_{i+1}, \dots, I_n$ of the higher strata. Formally, let

$$\Delta_i = (edb(P_i) \cup idb(P_i)) \cap (edb(P_{i+1}) \cup \dots \cup edb(P_n))$$

We must ensure that

$$\{p(\bar{t}) \in [\![P_i]\!]_{I_i} \mid p \in \Delta_i\} = \{p(\bar{t}) \in I_{i+1} \cup \dots \cup I_n \mid p \in \Delta_i\}$$

Key Steps. The algorithm first partitions P into strata $P_1, \dots P_n$. The strata can be computed using the predicates' dependency graph; see [88]. For each stratum P_i, it maintains a set of inputs $\mathcal{F}_i$, which contains inputs for P_i for which the algorithm failed to synthesize inputs for the lower strata $P_1, \dots, P_{i-1}$. We call the sets $\mathcal{F}_i$ *failed* inputs. All $\mathcal{F}_i$ are initially empty.

In each iteration of the while loop, the algorithm attempts to generate an input I_i for stratum P_i. At line 4, the algorithm checks whether $\mathcal{F}_i$ has exceeded a pre-defined bound $bound_{\mathcal{F}}$. If the bound is exceeded, it adds I_{i+1} to the failed inputs $\mathcal{F}_{i+1}$, re-initializes $\mathcal{F}_i$ to the empty set, and backtracks to a higher stratum by incrementing

Algorithm 2: Stratified Datalog input synthesis algorithm $\mathcal{S}_{Strat}$

```
   Input: Stratified Datalog program $P = P_1 \cup \cdots \cup P_n$, constraint $\varphi$ over $P_n$
   Output: An input $I$ such that $[[P]]_I \vDash \varphi$ or $\bot$
1  begin
2      $\mathcal{F}_1 \leftarrow \emptyset, \ldots, \mathcal{F}_n \leftarrow \emptyset; I_1 \leftarrow \bot, \ldots, I_n \leftarrow \bot; i \leftarrow n$
3      while $i > 0$ do
4          if $|\mathcal{F}_i| > bound_{\mathcal{F}}$ then
5              $\mathcal{F}_i \leftarrow \emptyset; \mathcal{F}_{i+1} \leftarrow \mathcal{F}_{i+1} \cup \{I_{i+1}\}$
6              $i \leftarrow i + 1;$   // backtrack to higher stratum
7              continue
8          $\psi_{\mathcal{F}} \leftarrow \bigwedge_{I' \in \mathcal{F}_i} \big(\neg \bigwedge_{p \in edb(P_i)} \text{EncodePred}(I', p)\big)$
9          if $i = n$ then
10             $\psi_i \leftarrow \varphi$
11         else
12             $\psi_i \leftarrow \bigwedge_{p \in \Delta_i} \text{EncodePred}(I_{i+1} \cup \cdots \cup I_n, p)$
13             where $\Delta_i = (edb(P_i) \cup idb(P_i)) \cap (edb(P_{i+1}) \cup \cdots \cup edb(P_n))$
14         $I_i = \mathcal{S}_{SemiPos}(P_i, \psi_i \wedge \psi_{\mathcal{F}})$
15         if $I_i \neq \bot$ then
16             $i \leftarrow i - 1$
17         else
18             if $i < n$ then
19                 $\mathcal{F}_i \leftarrow \emptyset; \mathcal{F}_{i+1} \leftarrow \mathcal{F}_{i+1} \cup \{I_{i+1}\}$
20                 $i \leftarrow i + 1$   // backtrack to higher stratum
21             else
22                 return $\bot$
23     return $I = \{p(\bar{t}) \in I_1 \cup \cdots \cup I_n \mid p \in edb(P)\}$
```

i. This avoids exhaustively searching through all inputs to find an input compatible with those synthesized for the higher strata.

At line 8, the algorithm uses the helper function $\text{EncodePred}(I', p)$. This function returns the constraint $\forall \overline{X}. \left(\bigvee_{p(\bar{t}) \in I'} \overline{X} = \bar{t}\right) \Leftrightarrow p(\overline{X})$, which is satisfied by an interpretation I iff I contains identical $p(\bar{t})$ predicates as those in I'. That is, if $I \vDash \text{EncodePred}(I', p)$ then for any $p(\bar{t})$ we have $p(\bar{t}) \in I$ iff $p(\bar{t}) \in I'$. Therefore, the constraint $\psi_{\mathcal{F}}$ constructed at line 8 is satisfied by an input I_i iff $I_i \notin \mathcal{F}_i$, which avoids synthesizing inputs from the set of failed inputs.

The constraint ψ_i in the algorithm constrains the fixed point of P_i. For the highest stratum P_n, ψ_i is set to the constraint φ given as input to the algorithm. For the remaining strata P_i, ψ_i is satisfied iff the fixed point of P_i is compatible with the synthesized inputs for the higher strata $P_{i+1}, \dots, P_n$. In addition to constraining P_i's *idb* predicates, we also constraint the input *edb* predicates. This is necessary to eagerly constrain the inputs.

At line 14, the algorithm invokes $\mathcal{S}_{SemiPos}$ to generate an input I_i such that $[[P_i]]_{I_i} \models \varphi_i \wedge \psi_{\mathcal{F}}$. The algorithm proceeds to the lower stratum if such an input is found ($I \neq \bot$); otherwise, if $i < n$ the algorithm backtracks to the higher stratum by increasing i and updating the sets $\mathcal{F}_{i+1}$, and if $i = n$ if returns $\bot$.

Finally, the while-loop terminates when the inputs of all strata have been generated. The algorithm constructs and returns the input I for P.

Theorem 3.4.6. Let P be a stratified Datalog program with strata $P_1, \dots, P_n$, and φ a constraint over predicates in P_n. If $\mathcal{S}_{Strat}(P, \varphi) = I$ then $[[P]]_I \models \varphi$.

Next, we prove the correctness of the stratified Datalog input synthesis algorithm $\mathcal{S}_{Strat}$, which uses the $\mathcal{S}_{SemiPos}$ algorithm as a building block.

Proof. By induction on the computation of the inputs $I_n, I_{n-1}, \dots, I_1$, we show that $[[P_i \cup \dots \cup P_n]]_{inp(P_i \cup \dots \cup P_n, I_i \cup \dots \cup I_n)} \models \varphi$ holds for $1 \leq i \leq n$. Note that the case for $i = 1$ proves the theorem.

Base Case: For the base case, we have $i = n$. Then $I_n = \mathcal{S}_{SemiPos}(P_n, \varphi)$. We have $inp(P_n, I_n) = I_n$, and by Theorem 1, we get $[[P_n]]_{I_n} \models \varphi$.

Inductive Step: For our inductive step, assume that $[[P_j \cup \dots \cup P_n]]_{inp(P_j \cup \dots \cup P_n, I_j \cup \dots \cup I_n)} \models \varphi$ holds for $i \leq j \leq n$, for some $1 < i \leq n$. We need to show that $[[P_{i-1} \cup \dots \cup P_n]]_{inp(P_{i-1} \cup \dots \cup P_n, I_{i-1} \cup \dots \cup I_n)} \models \varphi$. Recall that according to the semantics of stratified Datalog, the model $[[P_{i-1} \cup \dots \cup P_n]]_{inp(P_{i-1} \cup \dots \cup P_n, I_{i-1} \cup \dots \cup I_n)}$ is computed by first computing $[[P_{i-1}]]_{I_{i-1}}$ and then computing $[[P_i \cup \dots \cup P_n]]_I$ where I contains all ground atoms in $[[P_{i-1}]]_{I_{i-1}}$ together with ground atoms in $I' = inp(P_{i-1} \cup \dots \cup P_n, I_i \cup \dots \cup I_n)$. The only difference between I and $inp(P_i \cup \dots \cup P_n, I_i \cup \dots \cup I_n)$ therefore is that *edb* atoms of $P_i \cup \dots \cup P_n$ that are contained in $I_i \cup \dots I_n$ and are constructed using *idb* predicates of P_{i-1} are now derived by the the program P_{i-1} for the input I_{i-1}. The constraint φ_{i-1} constructed at line 12 of Algorithm 2 ensures that these two sets of ground atoms are identical. We can thus conclude that $[[P_{i-1} \cup P_i \dots \cup P_n]]_{inp(P_{i-1} \cup \dots \cup P_n, I_{i-1} \cup \dots \cup I_n)} \models \varphi$. □

3.5 NETWORK SYNTHESIS ALGORITHM

We now show how we use the algorithm $\mathcal{S}_{Strat}$ to synthesize network-wide configurations.

The main steps of our algorithm for synthesizing network configurations is given in Algorithm 3. Given a network specification $\mathcal{N}$, requirements φ_R, network topology T, and configuration constraints φ_C, we need to generate an input $\mathcal{C}$ (configuration) for $\mathcal{N}$ such that the configuration is well-formed $\mathcal{C} \models T \land \varphi_C$ and the network converges to a forwarding state compliant with the requirements $[\![\mathcal{N}]\!]_{\mathcal{C}} \models \varphi_R$. This is equivalent to synthesizing an input $\mathcal{C}$ such that $[\![\mathcal{N}]\!]_{\mathcal{C}} \models T \land \varphi_C \land \varphi_R$. Since $\mathcal{S}_{Strat}$ requires that the input constraint is defined over predicates in $\mathcal{N}$'s highest stratum, and $\mathcal{N}$ and $\mathcal{C}$ may refer to predicates in lower strata, as a first step we translate the constraint $T \land \varphi_C \land \varphi_R$ into a set Q of Datalog rules that contains a designated predicate $a_{\checkmark}$ such that for any input $\mathcal{C}$ for Q, we have $[\![Q]\!]_{\mathcal{C}} \models T \land \varphi_C \land \varphi_R$ if and only if $a_{\checkmark} \in [\![Q]\!]_{\mathcal{C}}$. This translation is denoted by $\overset{a_{\checkmark}}{\hookrightarrow}$ in Algorithm 3. We remark that the translation $\overset{a_{\checkmark}}{\hookrightarrow}$ is analogous to a standard query satisfiability reduction for Datalog.

Algorithm 3: The algorithm $\mathcal{S}_{Net}$ for synthesizing correct network-wide configurations.

```
Input: Network specification $\mathcal{N}$, global requirements $\varphi_R$, network topology $T$, and
       protocol configuration constraints $\varphi_C$
Output: An input $\mathcal{C}$ such that $\mathcal{C} \models T \land \varphi_C$ and $[\![\mathcal{N}]\!]_{\mathcal{C}} \models \varphi_R$, or $\bot$
1 begin
2     Compute $Q$ such that $T \land \varphi_C \land \varphi_R \overset{a_{\checkmark}}{\hookrightarrow} Q$
3     Stratify $\mathcal{N} \cup Q$ into $P$ with partitions $P_1, \dots, P_n$ such that $a_{\checkmark} \in idb(P_n)$
4     $I \leftarrow \mathcal{S}_{Strat}(P, a_{\checkmark})$
5     return $I$
```

Second, the algorithm extend the network specification $\mathcal{N}$ with the set Q of Datalog rules, which is obtained after translating $\varphi_R \land T \land \varphi_C$, and stratifies the rules in $\mathcal{N} \cup Q$ into stratified Datalog program P with the stratra $P_1, \dots, P_n$. Note that since the atom $a_{\checkmark}$ does not appear in the body of a rule in P, the rule with the head $a_{\checkmark}$ can be placed in the highest stratum P_n. Note that $edb(\mathcal{N}) = edb(P)$, and therefore extending $\mathcal{N}$ with Q does not change the signature of $\mathcal{N}$'s inputs.

Finally, we invoke the algorithm $\mathcal{S}_{Strat}$ for the inputs P and the constraint $a_{\checkmark}$ and the algorithm returns the answer output by the algorithm $\mathcal{S}_{Strat}$.

3.6 IMPLEMENTATION AND EVALUATION

In this section we first describe SyNET, and end-to-end implementation of our Datalog's input synthesis algorithm applied to the network-wide configurations synthesis problem. We then turn to our evaluation of SyNET on practical topologies and requirements. Our results highlight the feasibility of network-wide configurations synthesis.

3.6.1 *Implementation*

Our system SyNET is implemented in Python and automatically encodes stratified Datalog programs specified in the LogicBlox language [90] into SMT constraints specified in the SMT-LIB v2 format [91]. It uses the Python API of Z3 [92] to check whether the generated SMT constraints are satisfiable and to obtain a model.

SyNET supports routers that run both, OSPF and BGP protocols, and that can be configured with static routes. SyNET can support additional protocols if the stratified Datalog specifications of the protocols is supplied as input. SyNET uses natural splitting for routing protocols: external routes are handled by BGP, while internal routes are handled by IGP protocols (OSPF and static, where static routes are preferred over OSPF). We have partitioned the Datalog rules that capture these protocols and their dependencies into 8 strata. SyNET relies on additional SMT constraints to ensure the well-formedness of the OSPF, BGP, and static route configurations output by our synthesizer. For most topologies and requirements, the Datalog program reaches a fixed point within 20 iterations, and so we fixed the unroll and backtracking bounds ($bound_k$ and $bound_{\mathcal{F}}$) to 20.

SyNET is vendor agnostic with respect to the synthesized configurations. A simple script can be used to convert the output of SyNET into any vendor specific configuration format and then deploy them in production routers. Indeed, to test the correctness of SyNET, we implemented a small script to convert the input synthesized by SyNET to Cisco router configurations.

SyNET supports two key optimizations that improve its performance and are vital in making the synthesis algorithm applicable to practical network-wide configurations synthesis problems. The first optimization is *partial evaluation*: SyNET partially-evaluates Datalog rules with predicates whose truth values are known apriori. For example, all SetLink predicates are known and can be eliminated. This optimization reduces the number of variables in the rules and, in turn, in the generated SMT constraints. The second optimization is *network-specific constraints*: we have configured SyNET with generic constraints, which are true for all forwarding states, and with protocol-specific constraints, i.e., constraints that hold for any input to a

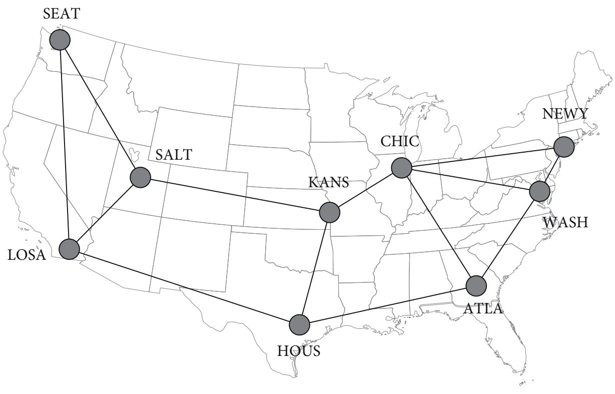


FIGURE 3.7: Internet2 topology.

particular protocol. An example constraint is: *"No packet is forwarded out of the router if the destination network is directly connected to the router"*. These constraints are not specific to particular requirements or topology. They are thus defined one time and can be used to synthesize configurations for any requirements and networks.

3.6.2 *Experiments*

To investigate SyNET's performance and scalability, we experimented with different: *(i)* topologies, *(ii)* requirements; and *(iii)* protocol combinations. Further to test correctness, we ran all synthesized configurations on an emulated environment of Cisco routers [93] and we verified that the forwarding paths computed match the requirements for each experiment.

Network Topologies. We used network topologies that have between 4 and 64 routers. The 4-router network is our overview example where we considered the same requirements as those described in Section 2.1. The 9-router network is Internet2 (see Figure 3.7), a US-based network that connects several major universities and research institutes. The remaining networks are $n \times n$ grids.

Routing Requirements. For each router and each traffic class, we generate a routing requirement that defines where the packets for that traffic class must be forwarded to. We consider 1, 5, and 10 traffic classes. For a topology with n routers and m traffic classes, we thus generate $n \times m$ requirements.

For topologies with multiple traffic classes, we add one external network announced by two randomly selected routers. We add requirements to enforce that all packets

		1 Traffic Class		5 Traffic Classes		10 Traffic Classes	
Protocol	# Routers	Avg	Std	Avg	Std	Avg	Std
Static	9	1.3s	(0.5)	2.0s	(0.1)	2.8s	(0.4)
	9 (Internet2)	1.3s	(0.5)	2.0s	(0.0)	4.0s	(0.8)
	16	5.9s	(0.3)	7.8s	(0.4)	11.2s	(0.4)
	25	32.0s	(0.6)	37.0s	(0.6)	46.1s	(0.9)
	36	2m49.7s	(3.0)	3m1.5s	(4.5)	3*m*27.0s	(4.4)
	49	12m29.2s	(7.0)	13m02.3s	(10.6)	14m10.7s	(15.0)
	64	46m36.2s	(49.0)	47m23.8s	(27.2)	49m22.2s	(39.3)
OSPF+Static	9	9.4s	(0.5)	19.8s	(0.4)	39.9s	(0.5)
	9 (Internet2)	9.0s	(1.4)	21.3s	(1.2)	49.3s	(0.5)
	16	43.5s	(0.7)	1m19.8s	(0.6)	4m5.8s	(1.6)
	25	2m55.2s	(6.1)	7m3.8s	(9.9)	15m56.4s	(38.1)
	36	10m00.5s	(9.5)	23m58.9s	(22.5)	1h11m38.2s	(127.5)
	49	24m11.6s	(43.5)	1h30m00.3s	(89.6)	5h22m55.8s	(421.2)
	64	2h22m13.2s	(209.9)	5h42m58.9s	(619.4)	21h13m16.0s	(1986.7)
BGP+OSPF+Static	9	15.3s	(0.5)	27.7s	(0.5)	1m0.5s	(2.6)
	9 (Internet2)	13.3s	(0.9)	22.7s	(0.9)	1m19.7s	(0.5)
	16	56.0s	(1.6)	2m24.7s	(0.9)	8m29.0s	(10.7)
	25	3m56.3s	(3.1)	8m46.3s	(5.3)	40m09.3s	(99.2)
	36	14m14.0s	(15.0)	43m38.0s	(5.7)	2h35m11.7s	(197.7)
	49	1h23m20.7s	(211.1)	2h15m18.0s	(12.8)	timeout (> 24h)	
	64	1h46m35.0s	(165.8)	7h24m51.3s	(519.2)	timeout (> 24h)	

TABLE 3.1: SyNET's synthesis times (averaged over 10 runs) for different number of routers, protocol combinations, and traffic classes.

destined to the external networks are forwarded to one of the two routers. This models a scenario where the operator is planning maintenance downtime for one of the two routers. Further, to show that SyNET synthesizes configurations with partially defined input and protocol dependencies, we assume the local BGP preferences are fixed by the network operator and thus SyNET has to synthesize correct OSPF costs to meet the BGP requirements.

Protocols. We consider three different combinations of protocols: *(i)* static routes; *(ii)* OSPF and static routes; and *(iii)* OSPF, BGP, and static routes. The protocol combinations *(i)* and *(ii)* ignore requirements for external networks since only BGP computes routes for them.

Experimental Setup. We run SyNET on a machine with 128GB of RAM and a modern 12-core dual-processors running at 2.3GHz.

Results. The synthesis times for the different networks and protocol combinations are shown in Table 3.1. Additionally, SyNET synthesizes the overview example's configuration described in Section 2.1 in 10 seconds. For the largest network (64 routers) and number of traffic classes (10 classes), SyNET synthesizes a configuration for static routes (protocol combination *(i)*) in less than 1h, and for the combination of static routes and OSPF, SyNET takes less than 22h. When using both OSPF and BGP protocols along with static routes, for all network topologies SyNET synthesizes configurations for 1 and 5 traffic classes within 8h; for 10 traffic classes, SyNET times out after 24h for the largest topologies with 49 and 64 routers.

Interpretation. Our results show that SyNET scales to real-world networks. Indeed, a longitudinal analysis of more than 260 production networks [94] revealed that 56% of them have less than 32 routers. SyNET would synthesize configurations for such networks within one hour. SyNET also already supports a reasonable amount of traffic classes. According to a study on real-world enterprise and WAN networks [95], even large networks with 100,000s of IP prefixes in their forwarding tables usually see less than 15 traffic classes in total.

While SyNET can take more than 24 hours to synthesize a configuration for the largest networks (with all protocols activated and 10 traffic classes), we believe that this time can be reduced through divide-and-conquer. Real networks tend to be hierarchically organized around few regions (to ensure the scalability of the protocols [96]) whose configurations can be synthesized independently. We plan to explore the synthesis of such hierarchical configurations in future work.

3.7 SUMMARY

We formulated the network-wide configurations synthesis problem as a problem of finding inputs of a stratified Datalog program and presented a new stratified Datalog input synthesis algorithm to solve this challenge. Our algorithm is based on decomposing the Datalog rules into strata and iteratively synthesizing inputs for the individual strata using off-the-shelf SMT solvers. This problem formulation allowed us to solve a more general problem and our contributions apply to domains other than networks. However, this generalization, while powerful, prevented our algorithm from using more efficient network-specific optimizations. Thus, limiting the synthesis speed of our implementation.

Furthermore, we implemented our approach in a system called `SyNET` and showed that it scales to realistic network size using any combination of OSPF, BGP and static routes. Network operators can now express their global routing requirements and use `SyNET` to automatically obtain network-wide configurations which ensure that routers compute a compliant forwarding state.

4

NETWORK-WIDE CONFIGURATIONS SYNTHESIS WITH AUTOCOMPLETION

While SyNET is capable of synthesizing network-wide configurations from routing high-level requirements, network operators often need to adapt the existing configurations of a network to comply with changing routing policies. Evolving existing configurations, however, is a complex task as local router-level configuration changes can have unforeseen global effects. Not surprisingly, reconfiguring the network often leads to mistakes that result in network downtimes.

Network configurations synthesis frameworks promise to alleviate most of the operator's burdens by deriving *correct* configurations from high-level objectives. While promising, network operators can be still reluctant to use existing synthesis systems for at least three reasons: *(i) interpretability*: the synthesizer can produce configurations that differ wildly from manually provided ones, making it hard to understand what the resulting configurations do. Moreover, small policy changes can cause the synthesized configurations (or configuration templates in the case of PropaneAT [55]) to change radically; *(ii) protocol coverage*: existing systems [37, 55] are restricted to producing BGP-only configurations, while most networks rely on multiple routing protocols (e.g., to leverage OSPF's fast-convergence capabilities); and *(iii) scalability*: today's networks are growing in both the number of psychical nodes and the number of traffic classes and policies they implement. Any practical configurations synthesis framework should be able to meet the scalability of realistic networks.

In this chapter, we present a system, NetComplete, which addresses the above challenges with partial synthesis. Rather than synthesizing new configurations from scratch, NetComplete allows network operators to express their intents by sketching the parts of the existing configurations that should remain intact (capturing high-level insights) and "holes" represented with symbolic values which the synthesizer should instantiate (e.g., OSPF link weights and BGP import/export policies). NetComplete then autocompletes these "holes" such that the resulting configurations lead to a network that exhibits the required behavior. Our approach supports a practical and relevant scenarios as few network operators ever start from scratch but rather modify their existing configurations (e.g., OSPF weights) to handle new routing requirements. This evolving approach also has the benefit of better explainability as large parts of the existing configurations are preserved in the newly synthesized configurations. Further, because we focus on synthesizing parts of the configurations, there is an opportunity to scale the synthesizer to realistic networks. This opportunity arises even though `NetComplete` is quite expressive: it handles static routes, OSPF, and BGP as well as a variety of essential routing requirements such as waypointing, failure-resilience, load-balancing, and traffic isolation.

`NetComplete` reduces the autocompletion problem to a constraint satisfaction problem that it solves with off-the-shelf SMT solvers (e.g., Z3 [92]). The main challenge is that a naive encoding of the problem leads to complex constraints that cannot be solved in reasonable time (e.g., within a day). To scale, `NetComplete` relies on two key insights: *(i)* partial evaluation along with *(ii)* network-specific heuristics to efficiently navigate the search space. Specifically, it speeds up BGP synthesis by propagating symbolic announcements through partial BGP policies allowing it to eliminate many variables. For OSPF, `NetComplete` is 100x faster than a naive encoding via a new counter-example guided inductive synthesis algorithm. Our evaluation shows that `NetComplete` autocompletes configurations for networks with up to 200 routers in a few minutes.

In this chapter, we start by motivating the need for partial configurations synthesis through practical scenarios in Section 4.1. Then, in Section 4.2, we present a motivating example of how a network operator would use a partial configurations synthesis framework, before diving into the details of our BGP and OSPF synthesis procedures in Section 4.3 and Section 4.4 respectively. Finally, we present our `NetComplete`'s implementation and experimental evaluation in Section 4.5.

4.1 MOTIVATING SCENARIOS

In this section, we motivate the need for `NetComplete` through *three practical use cases* rooted within existing network management practices. These use cases are difficult or practically impossible to solve today.

Scenario 1: *Evolving configurations preserving existing semantics*

Existing configurations typically embed deep knowledge of semantics and design guidelines. For instance, network operators often use specific OSPF weights to identify primary/backup links, and specific BGP local-preferences or communities to identify their peers. This (often unwritten) semantic helps them reason about the network-wide configurations. At the same time, these rules also reduce the operators' flexibility as it can complexify the implementation of new routing requirements, e.g., by requiring the modification of multiple weights instead of one.

`NetComplete` allows operators to communicate such semantics as constraints on the configurations sketch and let the synthesizer find a valid network-wide configurations that adheres to the operators' style.

Scenario 2: *Simplifying federated or constrained management*
Network configurations are often maintained by multiple teams of operators [97, 98], each responsible for some parts (e.g., edge vs core) or functionalities. Coordinating changes in these federated configurations tends to be challenging as multiple teams need to come together. With `NetComplete`, the operators can easily explore whether there is a way to implement the policy locally, for instance, without adapting the BGP configuration (i.e., by restricting changes to the OSPF configuration). Similar requirements appear in heterogeneous networks where not all routers support all protocols (e.g., due to licensing issues or device capabilities).

`NetComplete` allows operators to simply communicate such constraints as part of the sketch and let the synthesizer find a multi-protocol configuration.

Scenario 3: *Configuration Refactoring and Network Merging*
Network configurations evolve over time and this increases their complexity. Design decisions that made sense in the past may no longer do, requiring refactoring. Other examples calling for large refactoring include merging and acquisitions; e.g., when a company buys another one and wishes to integrate their networks [99].

`NetComplete` helps operators to refactor configurations by enabling them to morph entire pieces of their existing configurations, e.g., to adopt the configuration guidelines of one network and let the synthesizer compute and propagate the changes network-wide.

4.2 OVERVIEW

In this section, show how given a network topology, high-level routing requirements, and partial configurations, `NetComplete` autocompletes the partial configurations to correct network-wide configurations. First, we present a small running example and define `NetComplete`'s inputs. We then present the key synthesis steps to produce the output configuration before explaining the more complex steps in detail in the following sections.

4.2.1 *Running Example*

In Figure 4.2, we show how a network operator would use `NetComplete` to synthesize a network-wide configuration that enforces routing requirements. We consider that the AS number of the operator's network is $AS500$ and consists of four routers: A, B,

C, and D. Furthermore, this network is connected to one customer peer $AS100$ and three external peers: $AS200$, $AS300$, and $AS400$.

High-level Routing Policy. The policy for our example is given in Figure 4.1. This policy specifies the following routing behavior. Rule (1) disallows transit traffic between external peers; e.g., $AS200$ cannot send traffic to $AS300$ through the network. Rule (2) defines how the customer peer accesses prefixes announced by external peers: $AS300$ is most preferred, followed by $AS400$, and then $AS200$. Traffic to $AS200$ may exit via B or C, where B is preferred. Rules (3) and (4) capture traffic engineering requirements. Note that this policy can be formalized in a high-level network policy language language, such as Propane [37], Genesis [38], Frenetic [14], or `SyNET` (presented in Chapter 3).

4.2.2 `NetComplete` *Inputs*

`NetComplete` takes three inputs: *(i)* network topology, *(ii)* routing requirements, and *(iii)* configurations sketch.

(1) Network Topology. The network topology is given via a graph over the set of routers (A, B, C, and D) and external peers ($AS100$, $AS200$, $AS300$, and $AS400$). An edge represents a physical link that connects two nodes.

(2) Routing Requirements. We now describe the type of requirements supported by `NetComplete`. We start with some basic notation. A *routing path* is of the form: $P ::= Src \to R_1 \to \cdots \to R_n \to Dst$, where Src and Dst are source and destination routers, respectively, and $R_1, \ldots, R_n$ are router identifiers. We use a wildcard notation to denote sets of simple paths, i.e., paths without repeated nodes. For example, $Src \to * \to Dst$ denotes all simple paths from Src to Dst.

`NetComplete` supports positive and negative requirements. Positive requirements have the form:

$$Req ::= (P, \cdots, P) \mid (P = \cdots = P) \mid Req \gg Req$$

where P is a routing path. All routing paths that appear in a requirement must have identical source and destination. The semantics of requirements is as follows:

- An *any-path* requirement $\{P_1, \ldots, P_k\}$ is satisfied if the traffic from the source to the destination is forwarded along *any* available path in the given set. The requirement is not-applicable if all paths $P_1, \ldots, P_k$ are unavailable. We denote any-path requirement that includes *all* the paths between Src and Dst as $Src \to * \to Dst$. A *simple* requirement is an any-path requirement consisting of a single path. We remark that any-path requirements are used to ensure failure-resilience.

Rule 1 No transit between $AS200$, $AS300$, and $AS400$;
Rule 2 Traffic from the customer peer $AS100$ to the external peers prefers exit routers in order: $AS300$, $AS400$, $AS200$ via B, $AS200$ via C;
Rule 3 Traffic from $AS100$ to $AS300$ is load-balanced along $A \rightarrow C$ and $A \rightarrow D \rightarrow C$; if both paths are unavailable, then the path $A \rightarrow B \rightarrow C$ is used;
Rule 4 Traffic from $AS100$ to $AS400$ must follow the path $A \rightarrow B \rightarrow C$.

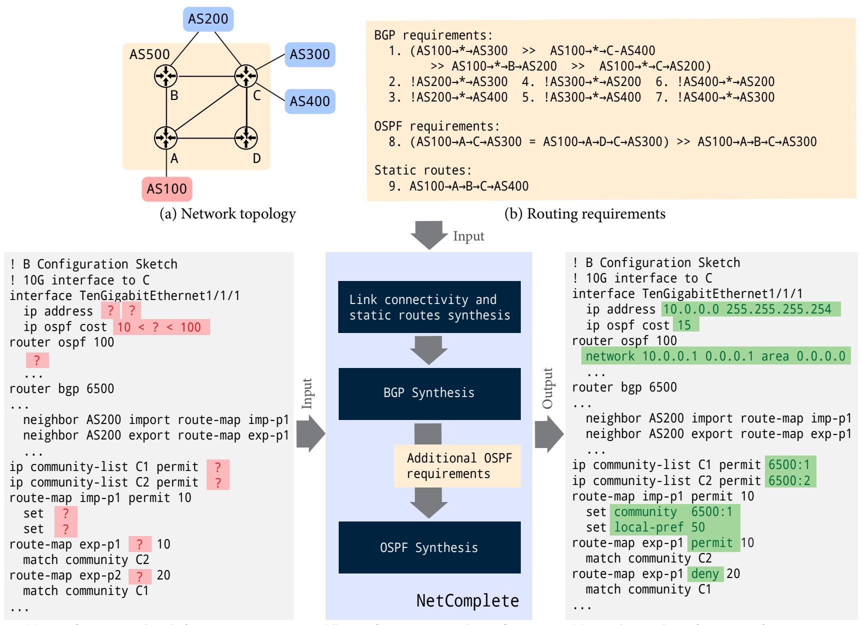
FIGURE 4.1: High-level policy for our running example.

- An *ECMP* requirement $(P_1 = \cdots = P_k)$ is satisfied if the traffic from *Src* to *Dst* is load-balanced among all available paths in the set $\{P_1, \ldots, P_k\}$. Note, this requirement is not-applicable if all paths $P_1, \ldots, P_k$ are unavailable. We remark that ECMP requirements are useful to capture load-balancing.
- An *ordered* requirement $Req_1 \gg Req_2$ defines a *preference* over requirements. This requirement is satisfied if the most preferred applicable requirement is satisfied, and it is not-applicable if both requirements are not-applicable. For example:

  $$(AS100 \rightarrow A \rightarrow B \rightarrow C \rightarrow AS300) \gg (AS100 \rightarrow A \rightarrow C \rightarrow AS300)$$

  is satisfied if traffic from $AS100$ to $AS300$ is forwarded along this path if it is available: $AS100 \rightarrow A \rightarrow B \rightarrow C \rightarrow AS300$. Otherwise, traffic is forwarded along the path: $AS100 \rightarrow A \rightarrow C \rightarrow AS300$.
- *Negative requirements* of the form $!\{P_1, \ldots, P_k\}$, where $\{P_1, \ldots, P_k\}$ is a set of routing paths. This requirement is satisfied if traffic is not forwarded along any path in this set. Negative requirements are useful to express traffic isolation.

The requirements for our example are given in Figure 4.2b. Policy rules 1, 2, 3, and 4, given in Figure 4.1, are specified as requirements 1, 2 – 7, 8, and 9, respectively. We use a natural assignment of requirements to protocols. Specifically, requirements 1 – 7 pertain to external peers and they are assigned to BGP. Requirement 8 pertains to traffic engineering within the network and is assigned to OSPF, which forwards traffic along the shortest path. Note that requirements 8 and 9 cannot be both enforced by OSPF. To enforce requirement 9, the cost of $A \rightarrow B \rightarrow C$ must be lower than that of $A \rightarrow C$ and $A \rightarrow D \rightarrow C$. However, this would also divert traffic from $AS100$ to $AS300$ to be forwarded along routers $A \rightarrow B \rightarrow C$, which would violate requirement 8. To this end, requirement 9 is enforced using a static route.



```
BGP requirements:
  1. (AS100→*→AS300  >>  AS100→*→C→AS400
        >> AS100→*→B→AS200  >>  AS100→*→C→AS200)
  2. !AS200→*→AS300  4. !AS300→*→AS200  6. !AS400→*→AS200
  3. !AS200→*→AS400  5. !AS300→*→AS400  7. !AS400→*→AS300

OSPF requirements:
  8. (AS100→A→C→AS300 = AS100→A→D→C→AS300) >> AS100→A→B→C→AS300

Static routes:
  9. AS100→A→B→C→AS400
```

```
! B Configuration Sketch
! 10G interface to C
interface TenGigabitEthernet1/1/1
  ip address ? ?
  ip ospf cost 10 < ? < 100
router ospf 100
  ?
  ...
router bgp 6500
...
  neighbor AS200 import route-map imp-p1
  neighbor AS200 export route-map exp-p1
  ...
ip community-list C1 permit ?
ip community-list C2 permit ?
route-map imp-p1 permit 10
  set ?
  set ?
route-map exp-p1 ? 10
  match community C2
route-map exp-p2 ? 20
  match community C1
...
```

```
! B Configuration Sketch
! 10G interface to C
interface TenGigabitEthernet1/1/1
  ip address 10.0.0.0 255.255.255.254
  ip ospf cost 15
router ospf 100
  network 10.0.0.1 0.0.0.1 area 0.0.0.0
  ...
router bgp 6500
...
  neighbor AS200 import route-map imp-p1
  neighbor AS200 export route-map exp-p1
  ...
ip community-list C1 permit 6500:1
ip community-list C2 permit 6500:2
route-map imp-p1 permit 10
  set community 6500:1
  set local-pref 50
route-map exp-p1 permit 10
  match community C2
route-map exp-p1 deny 20
  match community C1
...
```

(a) Network topology (b) Routing requirements

(c) Configuration sketch for router B (d) Configuration synthesis flow (e) Synthesized configuration for router B

FIGURE 4.2: Overview of `NetComplete`. The inputs are: **(a)** network topology, **(b)** routing requirements, and **(c)** configurations sketch. The output is a configuration for each router; one example is shown in **(e)**.

We remark that the requirements above can be specified manually by the operator, or using existing systems [14, 37, 38, 55] that compile high-level policies to forwarding paths.

(3) Configurations Sketch. Configurations sketches are normal router configurations where some of the parameters are left symbolic. To specify symbolic values, the operator tags parts of the configurations with a question mark symbol ? (instead of writing concrete values). The symbol ? represents: *(i)* specific attributes (e.g., OSPF link cost, BGP local preferences[1]); or *(ii)* entire import/export policies, e.g., `match ?, action ?`.

As an example, we depict the sketch of router B's configuration in Figure 4.2c. We remark that operators can write additional constraints to restrict how `NetComplete` instantiates symbolic parameters. For example, the symbolic OSPF link cost in the sketch of router B is constrained to values between 10 and 100.

1 Except BGP AS numbers, which are assigned based on higher-level considerations that are not captured in the requirements.

This sketching language enables `NetComplete` to be used in different scenarios. For example, changes can be restricted to certain parts of the network [Scenario 2]. By leaving most of the configurations symbolic, an operator can explore a large range of possible configurations that implement a given set of requirements [Scenarios 1 and 3]. Moreover, an operator can also provide a fully concrete configuration to verify its correctness.

4.2.3 *Configurations Synthesis*

`NetComplete` synthesizes a network-wide configurations that enforces the requirements in three steps.

First, it synthesizes the sessions between routers that have a physical link between them and may be necessary to enforce the routing requirements. Further, it configures any static routes defined in the requirements. For example, for requirement $AS100 \rightarrow A \rightarrow B \rightarrow C \rightarrow AS400$, `NetComplete` establishes a session between $A - B$ and $B - C$, and configures static routes at A and B.

Second, `NetComplete` synthesizes router-level BGP configurations based on the BGP routing requirements. To this end, `NetComplete` computes a propagation graph that captures which BGP announcements are exchanged between the routers and in what order they must be selected. `NetComplete` then synthesizes BGP configurations that enforce the constructed propagation graph. We explain this step in detail in Section 4.3. Note that BGP may select routes based on path costs (computed by OSPF). Therefore, whenever this is necessary to enforce the requirements, the BGP synthesizer outputs additional OSPF requirements to be enforced by the OSPF synthesizer.

Third, `NetComplete` synthesizes OSPF costs that enforce all OSPF requirements. This is a well-known hard problem that is difficult to scale to large networks. We solve the problem in Section 4.4 via a novel counter-example guided inductive synthesis algorithm.

If all synthesis steps succeed, `NetComplete` outputs a configuration that is guaranteed to enforce the requirements. Otherwise, a counter-example is returned to indicate that the requirements cannot be enforced for the given inputs. Based on this counter-example, the network operator can modify the partial configuration (by making more parameters symbolic) or adapt the requirements. We present a detailed evaluation of `NetComplete` with practical topologies and requirements in Section 4.5.

4.3 BGP SYNTHESIS

We now present `NetComplete`'s BGP synthesizer which takes as input BGP requirements and a sketch of the desired output BGP configurations then computes router-level BGP policies. It also outputs a set of OSPF requirements (to be fed to `NetComplete`'s OSPF synthesizer) if the BGP requirements cannot be enforced by BGP policies alone.

To synthesize BGP configurations, `NetComplete` uses a two-step approach. First, `NetComplete` computes a BGP propagation graph which defines a correct propagation of BGP announcements. Second, `NetComplete` uses a user-provided configurations sketch in addition to the knowledge of how BGP protocol propagates announcements and the BGP route selection process to synthesize BGP policies that enforce the BGP propagation graph.

In this chapter, we present the construction of the BGP propagation graph which defines a correct propagation of BGP announcements in Section 4.3.1. We illustrate `NetComplete`'s BGP sketches in Section 4.3.2 and propagation of (symbolic) announcements over them in Section 4.3.3. Then, we present the encoding of the BGP route selection process in Section 4.3.4. Finally, in Section 4.3.5, we describe our BGP synthesis procedure.

4.3.1 *BGP Propagation Graph*

We present how `NetComplete` builds, for each prefix, a propagation graph that defines correct enforcement of the BGP routing requirements for that prefix. In more details, `NetComplete` first constructs a graph G_{ebgp} that only considers announcements learned over eBGP. Then, it refines G_{ebgp} into G_{bgp}, which also defines how announcements are propagated internally (using iBGP). In Figure 4.3, we illustrate the steps of constructing a BGP propagation graph using our running example.

Construct eBGP Propagation Graph. The graph G_{ebgp} contains one node for each private/public AS. For our example, G_{ebgp} has one AS, $AS500$, that is managed by the network operator using `NetComplete` and four public ones $AS100, \dots, AS400$; see Figure 4.3.

The graph G_{ebgp} has two kinds of labeled edges: *propagate* and *block* edges, labeled with the preference order over announcements and, respectively, announcements that must be dropped.

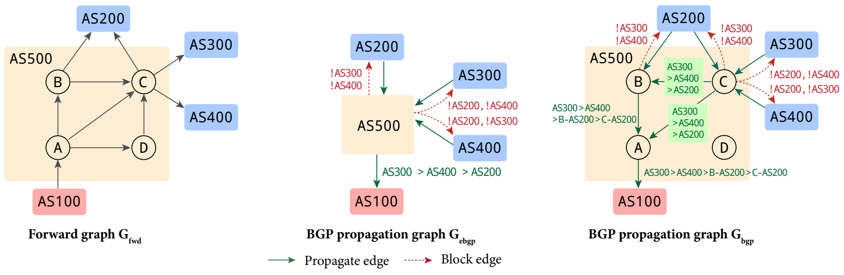
While the requirements define how the traffic should flow from the source to the destination (a destination router or AS), BGP propagates announcements

$$\begin{aligned} AS100 \to * \to AS300 &\gg AS100 \to * \to AS400 \\ &\gg AS100 \to * \to B \to AS200 \\ &\gg AS100 \to * \to C \to AS200 \end{aligned}$$

(a) Positive BGP requirements.

$$\begin{array}{lll} !AS200 \to * \to AS300 & !AS300 \to * \to AS200 & !AS400 \to * \to AS200 \\ !AS200 \to * \to AS400 & !AS300 \to * \to AS400 & !AS400 \to * \to AS300 \end{array}$$

(b) Negative BGP requirements.



(c) Step-by-step derivation of the BGP propagation graph.

FIGURE 4.3: Deriving a BGP propagation graph from BGP requirements and a network topology.

in the opposite direction. Thus, to add propagate edges, `NetComplete` traverses each positive BGP requirement backward and appends edges along the traversed ASes. For example, for the requirement $AS100 \to * \to AS300$ (see Figure 4.3a), `NetComplete` traverses three ASes and adds the propagate edges $AS300 \to AS500$ and $AS500 \to AS100$. While adding these edges, `NetComplete` tracks the set of symbolic announcements that must be propagated along them and labels the edges with the preference order based on the requirements.

To add block edges, `NetComplete` traverses each negative requirement and adds block edges to enforce it. For example, given the requirement $!AS200 \to * \to AS300$, `NetComplete` adds the block edge $AS500 \to AS200$, labeled with $!AS300$ in Figure 4.3c, to enforce this requirement.

Once G_{ebgp} is entirely constructed, `NetComplete` checks if preferences over announcements are consistent. To illustrate, suppose $AS1$ must select announcements from $AS2$, and $AS2$ must select from $AS3$. Then, the preferences over announcements labeled along the edges $AS3 \rightarrow AS2$ and $AS2 \rightarrow AS1$ must match.

Construct iBGP Propagation Graph. Next, `NetComplete` refines G_{ebgp} into a detailed propagation graph, G_{bgp}, that also accounts for iBGP (only in the ASes managed by the operators using `NetComplete`).

First, for each private AS in G_{ebgp}, `NetComplete` adds to G_{bgp} all BGP-enabled routers within that AS. For our example, `NetComplete` adds the routers A, B, C, and D.

Second, `NetComplete` connects the neighbor routers between ASes that have an edge in G_{ebgp}. For example, for edge $AS200 \rightarrow AS500$ in G_{ebgp}, `NetComplete` adds the edges $AS200 \rightarrow B$ and $AS200 \rightarrow C$ to G_{bgp}.

Finally, `NetComplete` extends the paths learned via eBGP. Note that in iBGP routers will not export routes learned from another iBGP router. Similar to G_{ebgp}, nodes in G_{bgp} are labeled with the preferences over announcements and `NetComplete` check if the preferences over announcements are consistent.

4.3.2 *BGP Policies*

We, first, present the semantics of BGP policies then we show how an operator would provide a sketch of BGP policies in `NetComplete`. Finally, we show how `NetComplete` encodes BGP policy sketches into SMT formulas.

A BGP policy applies on a set of announcements and has a match expression followed by zero or more actions. The match expression is a boolean formula over the announcement's attributes. If the match expression holds for the input announcement, then the actions are executed which modify the announcement's attributes or drop the announcement. For example, the following policy:

```
BGPPolicy
  match next-hop AS200
  set local-pref 10
```

matches an announcement whose `NextHop` attribute is set to $AS200$ and sets the value of attribute `LocalPref` to 10.

Sketching BGP Policies. `NetComplete` allows network operators to define the policy sketch at three levels of details; *(i)* everything is concrete (no holes), *(ii)* define the types of matches and actions but leave the specific values empty (see Figure 4.4a), *(iii)* or leave the matches and actions as holes (see Figure 4.4b).

Encoding BGP Policy Sketches. We illustrate the encoding of BGP sketches using SMT constraints. Consider the following BGP sketch:

```
AttributesSketch
  match next-hop AS200
  set local-pref ? < 50
```

This sketch would match any announcement that has the value $AS200$ set for the next hop attribute. If an announcement is matched, this policy sets the local preference of the output announcement to a value that is yet to be synthesized by the BGP synthesizer. As defined by the sketch, this local preference value must be smaller than 50. Note that this BGP policy does not change the remaining attributes (as there are no further actions).

We encode this BGP sketch as follows:

$$\begin{aligned}
&\textbf{if } \texttt{NextHop}_{A_{in}} = AS200 \\
&\quad \textbf{then } \big((\texttt{LocalPref}_{A_{out}} = Var_1) \wedge (0 < Var_1 < 50) \\
&\qquad\qquad \wedge(\forall X \in Attrs \setminus \{\texttt{LocalPref}\}.\ X_{A_{out}} = X_{A_{in}})\big) \\
&\quad \textbf{else} \quad \forall X \in Attrs.\ X_{A_{out}} = X_{A_{in}} \\
&\text{where } Attrs = \{\texttt{NextHop}, \ldots\} \text{ and } Var_1 \text{ are fresh variables}
\end{aligned}$$

Here, the variable Var_1 represents the local preference value that will be set by the BGP policy. A_{in} represents the input announcement (before the BGP policy processes it) and A_{out} the output one. The constraint formalizes that only input announcements with next hop equal to $AS200$ are matched. For matched announcements, the *then* constraint encodes that the output announcement has a local preference set to Var_1, which is a value smaller than 50, and all remaining attributes are identical to those in the input announcement (and thus remain unchanged). Finally, the *else* constraint ensures that if an announcement is not matched (its local preference is not $AS200$), then all attributes remain unchanged.

In Section 4.3.5, we show how `NetComplete` synthesizes BGP policy and instantiates the symbolic values in the given sketch to enforce the BGP propagation graph.

4.3.3 *Processing Symbolic Announcements*

We present how `NetComplete` processes symbolic BGP announcements that are passing through the various BGP policy sketches in the network. Given an announcement A, we write $\texttt{attr}_A$ to denote the attribute `attr` of A. For instance, $\texttt{LocalPref}_A$ returns A's local preference. Each announcement attribute either has a concrete value, if its value is fixed by the partial configurations, or a symbolic value, if a correct concrete value is yet to be discovered by the BGP synthesizer.

```
AttributesSketch
  match next-hop AS200
  set local-pref ? < 50
```

(a) Attributes sketch S_{attr}

```
AbstractSketch
  match ?
  set ?
```

(b) Abstract sketch S_{abs}

FIGURE 4.4: Example of two BGP policy sketches.

We represent announcements symbolically as their attributes' values are constrained by the BGP policies, which are yet to be synthesized. In NetComplete, each announcement A is represented with a set of symbolic variables $\texttt{Prefix}_A, \ldots, \texttt{NextHop}_A$. The set of possible attribute values of A is captured by a conjunction of constraints over these variables. For example, the constraint:

$$(\texttt{NextHop}_A = AS200) \land (0 < \texttt{LocalPref}_A < 50)$$

captures all announcements whose next hop is $AS200$ and local preference is a positive integer smaller than 50.

In addition to the standard BGP attributes that are listed in Figure 1.2 we introduce two boolean variables: $\texttt{Permitted}_A$, which indicates whether the announcement A is dropped, and $\texttt{eBGP}_A$, which indicates whether A is sent via eBGP or iBGP.

Processing Announcements with Policy Sketches. A BGP policy sketch takes as input a symbolic announcement A_{in} (a set of constraints over A_{in}'s attributes) and outputs another symbolic announcement A_{out}. To compute the set of possible output announcements for a given input announcement, we take the conjunction of the BGP sketch constraints with the constraint that captures the set of possible concrete input announcements.

To illustrate this step, consider the input announcement $\texttt{NextHop}_{A_{in}} = AS200$ and the BGP sketch in Figure 4.4a. Since the NextHop attribute is concrete and equal to $AS200$, NetComplete knows that the input announcement would match this policy. Therefore, NetComplete captures the set of possible output announcements with the constraint:

$$(\texttt{LocalPref}_{A_{out}} = Var_1) \land (0 < Var_1 < 50) \land \ (\texttt{NextHop}_{A_{out}} = \texttt{NextHop}_{A_{in}}) \ \land \ \cdots$$

Namely, the local preference of the output announcement is set to the value of Var_1, which is constrained to positive values below 50 (to be synthesized by NetComplete), and all remaining attributes are identical to those in the input announcement (captured with equality constraints, such as $\texttt{NextHop}_{A_{out}} = \texttt{NextHop}_{A_{in}}$).

As another example, consider the input announcement $\texttt{NextHop}_{A_{in}} = Var_1$ where the NextHop attribute is symbolic. When evaluating this announcement with

$$
\begin{aligned}
&\mathit{PrefNoIGP}(A_1, A_2) \Leftrightarrow\\
&\texttt{// 0) A1 is received and A2 is dropped}\\
&(\texttt{permitted}_{A_1} \wedge \neg\texttt{permitted}_{A_2})\\
&\texttt{// 1) Higher local preference}\\
&\vee\big(\texttt{permitted}_{A_1} \wedge \texttt{permitted}_{A_2} \wedge (\texttt{LocalPref}_{A_1} > \texttt{LocalPref}_{A_2})\big)\\
&\texttt{// 2) Lower AS path length}\\
&\vee\big(\texttt{permitted}_{A_1} \wedge \texttt{permitted}_{A_2} \wedge (\texttt{LocalPref}_{A_1} = \texttt{LocalPref}_{A_2})\\
&\qquad \wedge (\texttt{AsPathLen}_{A_1} > \texttt{AsPathLen}_{A_2})\big)\\
&\vdots\\
&\texttt{// 5) Prefer routes learned over eBGP}\\
&\vee\big(\texttt{permitted}_{A_1} \wedge \texttt{permitted}_{A_2} \wedge (\texttt{LocalPref}_{A_1} = \texttt{LocalPref}_{A_2})\\
&\qquad \cdots \wedge (\texttt{eBGP}_{A_1} \wedge \neg\texttt{eBGP}_{A_2})\big)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Pref}(A_1, A_2) \Leftrightarrow \mathit{PrefNoIGP}(A_1, A_2) \vee\\
&\texttt{// 6) Lower IGP cost}\\
&\big(\texttt{permitted}_{A_1} \wedge \texttt{permitted}_{A_2} \wedge (\texttt{LocalPref}_{A_1} = \texttt{LocalPref}_{A_2})\\
&\qquad \cdots \wedge (\texttt{IGPCost}_{A_1} < \texttt{IGPCost}_{A_2})\big)
\end{aligned}
$$

FIGURE 4.5: SMT encoding of the BGP tie-breaking process of announcements carrying same prefix.

the BGP sketch in Figure 4.4a, `NetComplete` captures the set of possible output announcements with the following constraint:

$$
\begin{aligned}
&\textbf{if } \mathit{Var}_1 = \mathit{AS}200\\
&\quad \textbf{then } (\texttt{LocalPref}_{A_{out}} = \mathit{Var}_2) \wedge (0 < \mathit{Var}_2 < 50)\\
&\qquad\qquad \wedge (\texttt{NextHop}_{A_{out}} = \texttt{NextHop}_{A_{in}}) \wedge \cdots\\
&\quad \textbf{else } (\texttt{NextHop}_{A_{out}} = \texttt{NextHop}_{A_{in}}) \wedge \cdots
\end{aligned}
$$

This constraint is more complex because the result of the match expression depends on the symbolic next hop (Var_1). If the next hop is $\mathit{AS}200$, then the local preference is set to Var_2 and all remaining attributes remain unchanged. Otherwise, all attributes in the output announcement A_{out} are identical to those in the input announcement A_{in}.

4.3.4 *SMT Encoding of the BGP Selection Process*

When a BGP router receives different announcements for the same prefix, it uses the BGP's tie-breaking process to select the best route; see Section 1.4. We encode the selection process into two SMT predicates: $\mathit{PrefNoIGP}(A_1, A_2)$ and $\mathit{Pref}(A_1, A_2)$. For any two symbolic announcements $A1$ and $A2$ carrying the same prefix, the predicate $\mathit{PrefNoIGP}(A_1, A_2)$ holds if and only if A_1 is preferred over A_2 without considering the IGP costs of A_1 and A_2. In other words, steps 1 through 5 in the BGP tie-breaking process, defined in Section 1.4, are sufficient to break the tie break between $A1$ and $A2$. While the predicate $\mathit{Pref}(A_1, A_2)$ holds if and only if A_1 is preferred over A_2 with considering the IGP costs of the paths that A_1 and A_2 are learned over.

Both $\mathit{Pref}(A_1, A_2)$ and $\mathit{PrefNoIGP}(A_1, A_2)$ are defined as a disjunction over the different cases defined by the BGP selection process. First, if A_2 is dropped, then A_1 is selected as a best route. Second, if both announcements are permitted, the router selects A_1 over A_2 if A_1's local preference is lower than that of A_2. Analogously, the constraint encodes cases 3 – 5 described in Section 1.4. Note, for the SMT encoding to be correct, the disjunction formula checks for each step that the previous step was a tie.

4.3.5 *BGP Policy Synthesis*

We now describe how `NetComplete` synthesizes BGP policies from requirements and policy sketches.

Encoding Requirements. Suppose that a router receives multiple announcements $A_1, \ldots, A_n$ to the same prefix. The BGP propagation graph identifies a preference with which the announcements must be selected by the router. Suppose the router must select announcements A_1, A_2, and A_3 in the order $A_1 \gg A_2 \gg A_3$. We encode this requirement with the following constraint:

$$\mathit{Pref}(A_1, A_2) \wedge \mathit{Pref}(A_2, A_3) \wedge \big(\forall i \in [4, .., n].\mathit{Pref}(A_3, A_i)\big)$$

Note that simpler requirements that do not stipulate a particular order are a special case. For example, if a requirement stipulates that an announcement A_k is selected as the best route, the above constraint becomes:

$$\forall i \in [1, n].\ k \neq i \implies \mathit{Pref}(A_k, A_i)$$

Overall Synthesis Algorithm Putting all pieces together, the complete algorithm employed by `NetComplete` to synthesize concrete BGP policies is as follows:

Step 1 (Section 4.3.1): Construct a BGP propagation graph G_{bgp} from the given requirements and the network topology.

Step 2 (Section 4.3.2): Encode the routers' BGP policy sketches. The result is a constraint φ_S over variables $\overline{S}$. Each concrete instantiation of the variables S identifies concrete BGP policies.

Step 3 (Section 4.3.3): Declare symbolic variables $\overline{A}$ to represent all the announcements propagated through the BGP propagation graph. Propagate all symbolic announcements through the policy sketches. The result is an SMT constraint $\varphi_{announcements}$ over the variables $\overline{S}$ and $\overline{A}$.

Step 4 (Synthesis without additional IGP requirements): Encode the route selection process and the requirements with the BGP selection predicate *PrefNoIGP*, resulting in SMT constraints φ_{select} and φ_{req} over the variables $\overline{A}$. If a model of $\varphi_{select} \wedge \varphi_{req}$ exists, then derive concrete BGP policies and return; otherwise, go to Step 5.

Step 5 (Synthesis with additional IGP requirements): Find the unsatisfiable core of $\varphi_{select} \wedge \varphi_{req}$ and derive a set S of pairs (A_1, A_2) of announcements that *cannot* be correctly selected without considering their IGP costs. Modify the constraint to:

$$\Big(\bigwedge_{(A_1,A_2)\in S} \texttt{IGPCost}_{A_1} < \texttt{IGPCost}_{A_2} \Big) \Rightarrow \varphi_{select} \wedge \varphi_{req}$$

If a model of this constraint exists, then derive BGP policies, create IGP requirements from the set S, and return; otherwise, return that the requirements cannot be satisfied.

4.4 OSPF SYNTHESIS

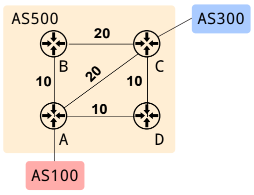
We now present `NetComplete`'s OSPF synthesizer. OSPF is a Dijskstra-based routing protocol that forwards traffic along the shortest path, where path costs are computed based on the OSPF cost (positive integer) attached to each link [32, 33]. `NetComplete` features a new *counter-example guided inductive synthesis* (CEGIS) [84] algorithm for OSPF that, given a set of OSPF requirements and a network topology, outputs OSPF link costs that enforce the requirements. Our algorithm can be tailored to support other Dijkstra-based routing protocols, such as IS-IS [34].

4.4.1 *SMT Encoding*

We phrase the OSPF synthesis problem as a constraint solving problem as follows: For any link that connects two nodes R to R' we introduce an integer variable $C_{R,R'}$ to represent the cost of link $R \to R'$. The cost of a path is given by the sum of the

$$\begin{aligned}&(AS100 \to A \to C \to AS300\\ &\qquad = AS100 \to A \to D \to C \to AS300)\\ &\gg AS100 \to A \to B \to C \to AS300\end{aligned}$$

(a) OSPF Requirements.



(b) Link costs.

FIGURE 4.6: Example of OSPF requirements and a correct link costs assignment.

link costs along that path. For example, the cost of $AS100 \to A \to B \to C \to AS300$, denoted by $Cost(A \to B \to C)$, is $C_{A,B} + C_{B,C}$. Note, the links $AS100 \to A$ and $C \to AS300$ are eBGP links and, hence, not included in the OSPF path cost. We also denote the *finite* set of all simple paths between two nodes R and R' with $Paths(R, R')$. We can encode that the path $P = AS100 \to A \to C \to AS300$ has the lowest cost among all other simple paths from $AS100$ to $AS300$ via:

$$\forall x \in Paths(AS100, AS300) \setminus \{P\}.\ Cost(A \to C) < Cost(x)$$

We can directly use this method to encode the enforcement of OSPF requirements; see Figure 4.6. For our example requirements, we obtain:

$$\begin{aligned}&Cost(A \to C) = Cost(A \to D \to C)\\ &\quad \land \big(Cost(A \to C) < Cost(A \to B \to C)\big)\\ &\quad \land \big(\forall x \in Paths(AS100, AS300) \setminus Reqs.\ Cost(A \to C) < Cost(x)\big),\\ &\text{where } Reqs = \{A \to C, A \to D \to C, A \to B \to C\}\end{aligned}$$

This constraint captures that: *(i)* $AS100 \to A \to C \to AS300$ and $AS100 \to A \to D \to C \to AS300$ must have equal costs, *(ii)* path $AS100 \to A \to C \to AS300$ has lower cost than $AS100 \to A \to B \to C \to AS300$, and *(iii)* all other paths have higher cost than $AS100 \to A \to B \to C \to AS300$. Note, in this example, $Paths(AS100, AS300) = S$. Therefore, we have $Paths(AS100, AS300) \setminus S = \varnothing$ and condition *(iii)* vacuously holds.

Naive OSPF Synthesis. A naive synthesis solution is to encode all requirements with constraints, as described above, and to then use a constraint solver to discover a model that identifies correct link costs. Unfortunately, phrasing the OSPF synthesis problem directly into SMT does not scale to large networks. The main issue is the for-all ($\forall$) quantifier in the constraints used to encode that a path has a lower cost among all other simple paths with the same source and destination. Note, counting all paths in a graph is ♯P-complete [100].

4.4.2 *Counter-Example Guided Inductive Synthesis for OSPF*

In this section, present our new counter-example guided inductive synthesis (CEGIS) algorithm for OSPF. CEGIS is a contemporary approach to synthesis, where a correct solution is iteratively learned from counter-examples [84]. CEGIS algorithms tend to work quite well in practice because often a small number of counter-examples (that is, few iterations) is sufficient to discover a correct solution.

The OSPF synthesis problem amounts to finding a model of logical constraints of the form:

$$\exists \overline{C}.\ \textsc{EncodeOSPF}(\overline{C}, r, \mathit{Paths}(r))$$

where $\overline{C}$ is the set of variables that represent link costs, r is an OSPF requirement, $\mathit{Paths}(r)$ is the set of all paths from the source *Src* and destination *Dst* provided in the requirement r, and $\textsc{EncodeOSPF}(\overline{C}, r, \mathit{Paths}(r))$ returns a logical formula that encodes the requirement's satisfaction (as described in Section 4.4.1). Finding a model of this formula directly using a constraint solver is difficult due to the large number of paths in $\mathit{Paths}(r)$. To avoid this quantifier, CEGIS restricts the constraint to a (small) set of paths $S = \{P_1, \dots, P_n\} \subseteq \mathit{Paths}(r)$. The resulting constraint is:

$$\exists \overline{C}.\ \textsc{EncodeOSPF}(\overline{C}, r, S)$$

which is easier to solve by existing constraint solvers. A model of this constraint identifies link costs that imply that the requirement holds over the paths in S. However, it may not hold over all paths in $\mathit{Paths}(r)$. The idea of CEGIS is to check the requirement over all paths and to obtain a concrete counter-example that violates it, if one exists; we remark that the step of checking is usually efficient. The set S is then iteratively expanded with counter-examples until a correct solution is found.

Algorithm We show the main steps of our CEGIS algorithm in Alg. 4. For each requirement $r \in \mathit{Reqs}$, the algorithm declares a set S_r (line 3). The algorithm then iteratively repeats the following steps. For each requirement $r \in \mathit{Reqs}$, the algorithm samples b paths from the source to the destination of the requirement r and adds these to S_r (line 7). It then encodes the requirement's satisfaction with respect to S_r (line 8) and conjoins the result to φ (line 9). If the resulting constraint φ is unsatisfiable, it means the requirements cannot be satisfied and the algorithm returns $\perp$ to indicate this. Otherwise, it obtains a model M of the constraints φ (line 12), which defines a concrete value for each link cost variable.

The algorithm then checks whether these costs defined by M enforce the requirements *Reqs* (over all paths). If the requirements are satisfied, the algorithm returns $M(\overline{C})$ (line 14), i.e. it returns the values associated to the link cost variables $\overline{C}$. Otherwise, it obtains a concrete counter-example as a pair (r, path) of a path *path*

Algorithm 4: CEGIS algorithm for synthesizing OSPF link costs with respect to OSPF requirements.

```
Input: OSPF requirements $Reqs = \bigcup_i r_i$, link cost variables $\overline{C}$, bound $b$
Output: OSPF link costs
1  begin
2      for $r \in Reqs$ do
3          $S_r = \emptyset$
4      while $true$ do
5          $\varphi = true$
6          for $r \in Reqs$ do
7              $S_r \leftarrow S_r \cup \textsc{SamplePaths}(r, b)$
8              $\varphi_r \leftarrow \textsc{EncodeOSPF}(\overline{C}, r, S_r)$
9              $\varphi \leftarrow \varphi \wedge \varphi_r$
10         if $\textit{Unsat}(\varphi)$ then
11             return $\bot$
12         $M \leftarrow \textsc{Model}(\varphi)$
13         if $\textit{CheckReqs}(M, Reqs)$ then
14             return $M(\overline{C})$
15         $(r, path) \leftarrow \textsc{CounterExample}(M, Reqs)$
16         $S_r \leftarrow S_r \cup \{path\}$
```

that violates a requirement r, and expands the set S_r with *path* (line 16). This ensures that the counter-example is avoided in the next iteraion. Further, to reach a solution faster, the algorithm samples additional b paths for each requirement r and adds them to S_r. These steps are repeated until a solution is found or the requirements are deemed unsatisfiable.

4.5 IMPLEMENTATION AND EVALUATION

We implemented `NetComplete` in about $10K$ lines of Python code using SMT-LIB v2 [91] and Z3 [92]. `NetComplete`'s SMT formulas use the theories of linear integer arithmetic and quantifier-free uninterpreted functions. Our prototype takes as input partial configurations (combining OSPF, BGP, and static routes) and outputs completed ones. We support standard Cisco commands for setting OSPF costs and BGP policies and can easily extend our code base to support other languages.

In this section, we show that `NetComplete`'s implementation is practical and scales to realistic networks. Specifically, we measure: *(i)* `NetComplete` OSPF and

BGP synthesis times in growing network topologies; *(ii)* the impact of having more or less symbolic variables in the sketches; and *(iii)* how `NetComplete` compares against competing approaches such as `SyNET`.

4.5.1 *Methodology and Datasets*

We evaluate `NetComplete` on real-world network topologies from the well-known dataset Topology Zoo [94]. However, this dataset provides only topologies without any high-level policies and routers configurations. In this section, we show our methodology of how we select our evaluation topologies from Topology Zoo and how we generate high-level routing requirements and configurations sketches for the selected topologies.

Topologies. We sample 15 network topologies from Topology Zoo that we classify according to the number of routers in the topology: *small* (from 32 to 34 routers), *medium* (from 68 to 74 routers), and *large* (from 145 to 197 routers). We select 5 topologies per category.

Requirements. For each chosen topology, we generate a set of routing requirements to evaluate our implementation. We generate four types of routing requirements (simple, any-path, ECMP, and ordered). For each requirement, we randomly select a source-destination pair (Src, Dst) from the routers in the topology. For simple path requirements, we choose a random feasible path to forward traffic from Src to Dst. For the other requirements, we first choose two paths P_1 and P_2 from Src to Dst and then we construct (P_1, P_2) for any-path requirements, $(P_1 = P_2)$ for ECMP, or $P_1 \gg P_2$ for ordered requirements. For each topology, we generate multiple sets of requirements of size 2, 8, and 16. We generate all four types of requirements for the OSPF evaluation, and only generate simple and ordered path requirements for the BGP evaluation. Indeed, any-path and ECMP requirements are typically internal requirements and are therefore typically enforced by IGP protocols.

Sketches. We construct configuration sketches for each topology from a fully concrete configuration (which we synthesize using `NetComplete`) for which we randomly make a given percentage of the variables symbolic. For instance, to generate partial OSPF (resp. BGP) configurations that are 50% symbolic, we randomly make 50% of the edges (resp. BGP import/export policies) in the synthesized concrete configurations symbolic.

Validation. We validate that our synthesized configurations comply with the corresponding requirements in an emulated environment composed of Cisco routers [93].

4.5.2 *Results*

We now present our experimental-evaluation results focusing first on OSPF synthesis, before then BGP synthesis, and finishing with a comparison with `SyNET`. We run all our experiments on a server with 128GB of RAM and a 12-core dual-processors running at 2.3GHz. Unless indicated, we report averaged results over 5 runs and across topologies of the same class.

OSPF Synthesis. We first illustrate the effectiveness of synthesizing OSPF configuration using our CEGIS algorithm versus a naive algorithm in which the entire $\exists\forall\varphi$ constraint, i.e., exploring all the paths between the source and destination in one shot, is directly fed to the solver. We then evaluate how sketches affect the overall synthesis time.

We report our results in Table 4.1 and convey four important insights. *First*, CEGIS significantly outperforms naive OSPF synthesis, especially in large networks where naive synthesis does not even terminate within a day. *Second*, we see that the synthesis time is proportional to both the topology size and the number of requirements. Indeed, the number of symbolic variables is equal to the number of symbolic edge costs, while the number of constraints is proportional to the requirements size and the number of available paths. *Third*, ordered path requirements take more time to synthesize than the other requirements. This is expected as such requirements specify a strict sequence of paths making the search space more sparse. *Fourth*, the use of more concrete values significantly reduces the synthesis time, especially for ordered path requirements with reductions up to 70%. We further illustrate this behavior in Figure 4.7 which depicts the times required to synthesize 16 ordered path requirements for the large networks as a function of the percentage of symbolic values. We see that `NetComplete` indeed leverages the concrete variables and the reduced search space to synthesize configurations faster.

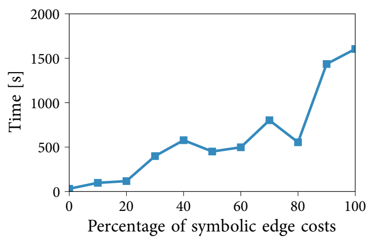


Figure 4.7: `NetComplete` synthesizes ordered path requirements faster when the configuration sketch provides more concrete values for edge costs.

		2 requirements				8 requirements				16 requirements			
Network	**Req.**	50% symbolic		100% symbolic		50% symbolic		100% symbolic		50% symbolic		100% symbolic	
size	**type**	**CEGIS**	**Naive**	**CEGIS**	**Naive**	**CEGIS**	**Naive**	**CEGIS**	**Naive**	**CEGIS**	**Naive**	**CEGIS**	**Naive**
Small	Simple	0.41	0.93	0.43	1.04	1.66	2.42	1.67	2.73	3.33	6.95	3.39	8.02
	Any-path	0.62	2.00	0.67	2.38	2.31	12.27	2.38	14.48	4.63	7.22	4.76	8.58
	ECMP	0.48	0.84	0.53	0.94	1.72	5.02	1.77	5.76	3.44	3.16	3.48	3.61
	Ordered	0.55	0.54	0.68	0.64	2.90	2.93	5.49	3.50	4.76	5.07	7.93	6.05
Medium	Simple	0.79	790.04	0.81	1554.81	3.06	19613.55	3.10	20.60	6.17	3238.46	6.18	6039.24
	Any-path	1.27	1677.30	1.28	4208.68	4.89	18758.02	4.94	66.10	9.70	107.13	9.83	122.68
	ECMP	0.85	567.02	0.86	1370.70	3.16	5643.60	3.24	22272.88	6.34	45.32	6.39	51.61
	Ordered	1.76	450.64	2.81	732.60	30.83	2942.83	33.60	8636.21	31.08	49.43	43.63	58.54
Large	Simple	1.78	> 24h	1.85	> 24h	7.35	> 24h	7.40	> 24h	13.90	> 24h	14.03	> 24h
	Any-path	4.23	> 24h	4.33	> 24h	16.59	> 24h	16.89	> 24h	32.61	> 24h	33.01	> 24h
	ECMP	1.83	> 24h	1.89	> 24h	7.07	> 24h	7.14	> 24h	13.37	> 24h	13.52	> 24h
	Ordered	6.90	> 24h	15.00	> 24h	33.81	> 24h	44.72	> 24h	249.48	> 24h	1155.19	> 24h

TABLE 4.1: Using Counter-Example Guided Inductive Synthesis (CEGIS) to synthesize OSPF weights is *considerably* faster than a naive OSPF algorithm which aims to solve all constraints at once.

BGP Synthesis. We now evaluate the effectiveness of our BGP synthesizer and how it leverages partial evaluation to concretize up to 25% of the symbolic variables and therefore speed up the overall synthesis time.

Topo	**Req.**	**Total #**		16 reqs.
	type	**of vars**	**Min % Eval**	**Max % Eval**
Small	Simple	58578	9.62%	18.76%
	Ordered	37662	16.75%	18.76%
Medium	Simple	98683	7.27%	13.54%
	Ordered	58924	10.02%	22.81%
Large	Simple	83832	11.93%	14.57%
	Ordered	29565	22.56%	25.07%

TABLE 4.2: Number of generated symbolic variables. Thanks to partial evaluation, `NetComplete` is able to evaluate between 7% and 25% of the symbolic variables—making BGP synthesis significantly faster.

In Table 4.2, we show the average number of generated symbolic variables for each group. We see that the number of generated symbolic variables is not directly related to the topology size and the number of requirements: the number of variables for medium topologies can exceed the ones of larger topologies. For BGP, the number of variables indeed depends on: *(i)* the number of routers (and their connectivity) in the computed BGP propagation graph; *(ii)* the complexity of the configuration sketch; and *(iii)* the effectiveness of partial evaluation.

Regarding partial evaluation, we observe that `NetComplete` manages to evaluate between 7% and 25% of the generated symbolic variables (see Table 4.2), which makes BGP synthesis proportionally faster. Indeed, in Figure 4.8, we show how the BGP synthesis time evolves linearly as a function of the number of symbolic variables. We also see that `NetComplete` always manages to synthesize BGP configurations in less than 14min.

Comparison to `SyNET` We now compare the synthesis time of `NetComplete` to `SyNET`. Specifically, we compare `NetComplete` and `SyNET` running times for the worst-case scenario reported in [101] involving 10 requirements defined in topologies with 49 and 64 routers. Since `SyNET`defines requirements in terms of the number of traffic classes and not forwarding paths as `NetComplete`, we first translate each traffic

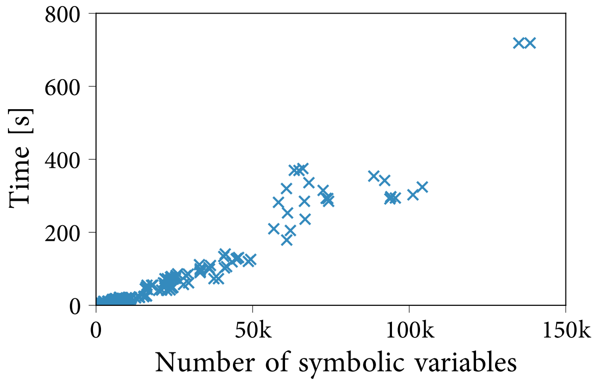


FIGURE 4.8: BGP synthesis time grows linearly with respect to the number of symbolic variables.

class to a set of simple path requirements. To ensure a fair comparison, we provide `NetComplete` with entirely symbolic sketch since `SyNET`does not accept sketches.

# Rtrs	Protocol	`SyNET`	`NetComplete`
49	Static	14m11s	0.05s
	Static + OSPF	5h22m56s	2m1s
	Static + OSPF + BGP	timeout (> 24h)	44m2s
64	Static	49m22s	0.06s
	Static + OSPF	21h13m16s	2m22s
	Static + OSPF + BGP	timeout (> 24h)	6h6m30s

FIGURE 4.9: `NetComplete` is > 600× faster than [101].

Our results, in Figure 4.9, shows that `NetComplete` is at least 600× faster than `SyNET` and is able to synthesize configurations for larger topologies that `SyNET` timed out on. This speed up stems from two factors. First, `NetComplete` does not use an SMT solver for the requirements that it can solve directly (such as synthesizing static routes). Second, `NetComplete` relies on domain-specific heuristics (CEGIS and partial evaluation) to reduce the search space, while `SyNET` relies on the generic optimizations of the underlying SMT solver.

4.6 SUMMARY

We presented NetComplete, the first scalable network-wide configuration synthesizer to support multiple protocols and a partial sketch of the desired configuration. While SyNET presented a general network-wide synthesis framework, NetComplete focused on the practical aspects of this problem. Namely, unlike SyNET, NetCompleteis capable of synthesizing configurations for a limited set of built-in protocols. However, this design choice enables NetCompleteto use more efficient optimization strategies that enables it to scale to support networks larger in size and to evolve existing configurations rather than synthesizing new ones.

NetComplete features a new BGP synthesis procedure that supports BGP configuration sketches and partial computations over symbolic announcements. It also introduces an efficient synthesis procedure for the widely-used OSPF protocol. This procedure is based on counter-example guided inductive synthesis and achieves significant speedups (> 100x) over existing solutions.

Finally, we presented a comprehensive set of experimental results, which demonstrate that NetComplete can autocomplete configurations for large networks with up to 200 routers within few minutes.

Part III

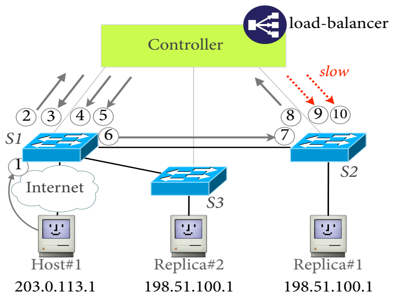
NETWORK VERIFICATION

5 SDN PROGRAMMING AND CONCURRENCY ISSUES

In the past few years, Software-Defined Networking (SDN) managed to establish itself as a promising approach for designing and operating computer networks. SDN's vision focuses on two fundamental premises: separating the control and data planes and centralizing the controller; see Section 1.5. However, realizing this vision in practice requires developers to build highly sophisticated and reliable *SDN controllers* (i.e., control software operating on top of a network). An SDN controller, at its core, is an event-driven program whose goal is to compute, maintain, and populate the forwarding table of each SDN switch in the network. SDN controllers operate in highly asynchronous environments where events such as packets arriving at a switch, link or node failures, or expiring flows can be dispatched to the controller at any time, all non-deterministically. The controller can request synchronously other events such as statistics from the switches. Building such highly asynchronous programs is known to be a challenging problem due to inadvertently introducing harmful concurrency errors.

In the context of SDN, there are two places where concurrent interference can occur: *(i)* within the SDN control software itself (e.g., if it is multi-threaded or distributed); and *(ii)* at the interface between the control software and the SDN switches. SDN switches can indeed be seen as memory locations which are *read* and *modified* by various events and entities. While the first kind of interference can be detected with standard approaches [102], the second kind of interference is harder to detect as it often depends on a particular ordering of specific, but unpredictable events. Yet, detecting these interferences is crucial as they are typically at the root of deeper semantic problems such as black holes, forwarding loops or non-deterministic forwarding.

In the following, we say that a *concurrency issue* arises when there are two unordered accesses to the switch flow table, one of which is a write produced by the controller. A *harmful concurrency violation* is a bug that affects the correctness of the SDN controller's execution [103].



(a) Events sequence

```
if dst==server:
   _rep=rep[idx]
   idx=(idx+1)%2
   install_path(src,_rep)
   install_path(_rep,src)
   packet_out(pkt,in_sw)

def install_path(s,d):
   path=dijkstra(s,d)
   for p in path:
      flow_mod(i,s,d,fwd(p[i+1]))
```

(b) Basic load-balancing application

FIGURE 5.1: An example of a simple load-balancing application and a sequence of events, which leads a concurrency violation that triggers a forwarding loop.

5.1 A NON-DETERMINISTIC FORWARDING LOOP IN A LOAD BALANCER

We start this section by introducing the example application showing a sequence of events that leads to a concurrency violation. Then, demonstrate how the concurrency violations leads to violations in high-level correctness properties.

Consider a simple SDN controller program which runs a load-balancer application (see Figure 5.1) that directs external requests to a chosen replica in a round-robin fashion. For this basic program, we show that a data race can happen; i.e., two unordered operations on a flow table, one of which is a flow table write event. A race condition can cause the traffic to be trapped in a forwarding loop or be delivered to different replicas.

Now, consider the following sequence of events: an external host, Host #1, sends a request directed to a farm of web server replicas identified by the IP address `198.51.100.1`. That request hits the first switch in the network, *S*1 ①; since it is a new request, *S*1 sends the request to the controller ② in an OpenFlow message called PACKET_IN. The controller *(i)* elects Replica #1 to serve the new request; *(ii)* computes the shortest-path between *S*1 and Replica #1 (as well as the return path) according to the load-balancing application defined in Figure 5.1b; *(iii)* pushes down two flow table write events (called FLOW_MODs), one for each traffic direction, on *S*1 ③–④ and similarly on *S*2 ⑨–⑩ to forward the packets from Host #1 to Replica #1 (as well as the return path); and *(iv)* sends the request back to *S*1 in a PACKET_OUT ⑤. *S*1 sends the request to *S*2 ⑥. In this trace, the packet hits *S*2 ⑦ *before* the corresponding flow rules ⑨–⑩ are installed on *S*2, causing the packet to be sent back to the controller ⑧ (as switch $S2$ does not have any entries installed to handle this packet). Assuming a round-robin selection algorithm, the controller assumes this is a new request and elects Replica #2, computes the shortest-path between $S2$ and Replica #2 and pushes down the corresponding flow rules on $S2$, $S1$, and $S3$; for brevity we do not show this step in Figure 5.1.

From this point on, the traffic is processed *incorrectly*, in a non-deterministic manner, as $S1$ and $S2$ each have forwarding entries with the same priority that match each direction of the traffic. Concretely, both directions of the traffic either end up caught in a forwarding loop, if $S1$ (respectively, $S2$) uses the rule to forward the traffic to $S2$ (respectively, $S3$), or hits one of the two replicas, non-deterministically. As replicas maintain state for each connection they receive, changing the replica on-the-fly will cause the connection to drop. In both cases, traffic ends up being lost.

Concurrency Violations. In this example, the concurrency error arises between the *read event* caused by the packet received by $S2$ ⑦ and the *write event* ⑨ matching it which leads to lost traffic. Note that the controller could prevent this problem by using OpenFlow Barrier messages (BARRIER_REQUEST and BARRIER_REPLY) to ensure the rules are installed on both $S1$ and $S2$ before pushing the request back to $S1$.

High-Level Properties Violations. In particular, we focus on two high-level properties: *update isolation* and *packet coherence*.

Informally, update isolation dictates that different policy changes do not interfere with each other (we provide a formal definition in Section 6.5.2). Note that update isolation property only pertains to concurrency violations between different flow table write events. In our example, the load-balancer application issued two policy changes: the first policy selects Replica #1, while the second policy selects Replica #2. However, due to the concurrency violations between the *read event* caused by the packet received by $S2$ ⑦ and the *write event* ⑨, the network violates the update isolation property. In particular, $S1$ forwards traffic according to the first policy (to Replica #1) while $S2$ is forwarding traffic according to the second policy (to Replica #2). Note that solving the underlying concurrency violation also solves the violation of this property.

On the other hand, the packet coherence property dictates that each packet is processed entirely by one consistent global policy (we provide a formal definition in Section 6.5.3). Note that update packet coherence property only pertains to concurrency violations between flow table write events and packets reading the flow table entries. In our example, the switches process the first packet of the request using two policies (using Replica #1 and Replica #2). In this instance, the violation of the packet coherence properties could lead to a forwarding loop in the network.

5.2 PROBLEM STATEMENT

A practical SDN concurrency analysis tool would help SDN controller developers not only to detect concurrency violations but also to provide useful insights into any violations of high-level properties. Informally, an SDN concurrency analysis tool takes as input an event trace π and reports to the developers *(i)* harmful concurrency violations; *(ii)* high-level properties violations to understand the broader impact of the harmful concurrency violations on the correctness of the SDN controller; and *(iii)* a small set of representative concurrency violations (ideally, one violation per bug). To obtain an event trace π, the developers could collect the events from their existing quality-assurance (QA) environment; in which the controller runs on real hardware or an emulated network to detect any bugs before deployment to production. However, even a small event trace could contain millions of events; depending on the size of the network, the number of packets flowing through the network, and the complexity of the SDN applications running inside the controller. Thus, a concurrency analysis tool could report thousands or even tens of thousands of concurrency violations overwhelming the developers, making such tools impractical. Thus, it is crucial for a practical concurrency analysis tool to report only harmful concurrency violations. Due to the nature of the network, the same bug that is the root cause of a concurrency violation could be triggered multiple times and report different violations. Hence, ideally, a concurrency analysis tool should report one representative violation per bug so the controller developer could focus on fixing a given bug. For instance, in our example load-balancer app, every new request triggers the same concurrency bug (see Section 5.1).

Definition 5.2.1. *SDN Concurrency Analysis Problem Statement*
Given an event trace π produced by an SDN network, find all the harmful concurrency violations and the high-level properties that are violated as a consequence.

5.3 OUR CONTRIBUTIONS

We next summarize our main contributions to address the SDN concurrency analysis problem:

1. A thorough happens-before model (HB model) [104] which precisely captures the asynchronous interaction between an OpenFlow-based SDN controller and the SDN switches (Section 6.3). The HB model is a relation that precisely captures the ordering of events in an SDN network based on the potential causality relation between them. We use the ordering captured by the HB model to detect any potential unordered memory access and, hence, potential

concurrency violations. Note, not all reports from this analysis alone are true violations that affect the correctness of the SDN network.

2. A set of effective filters that dramatically reduce the number of reports, including a *commutativity specification* which captures the precise conditions under which two operations on the network switch commute (Section 6.4). Together, the specification and the filters reduce the number of reported issues by several orders of magnitude.

3. A set of domain-specific features to measure the similarities between concurrency violations. We use the set of domain-specific features to cluster the related concurrency violations and rank the violations in each cluster. Then we develop a ranking algorithm to select a representative candidate of each cluster (Section 6.6). In a case study, we show that solving the concurrency bug using the representative violation caused 99.23% of the violations to disappear (Section 6.8).

4. A complete implementation of `SDNRacer`, a dynamic analyzer which can readily analyze production-grade (single and multi-threaded) SDN controllers for various properties including data races, packet coherence, and update isolation (Section 6.7).

5. A comprehensive evaluation of `SDNRacer` attesting that it can uncover harmful and previously unknown bugs in existing SDN applications [105–113] (Section 6.8).

5.4 RELATED WORK

Data Plane Verification. Several projects aim to verify the correctness of SDN networks. Anteater [114], HSA [115] and Libra [116] collect snapshots of the network forwarding state and check if it violates certain properties. VeriFlow [117] and NetPlumber [118] build on this by allowing real-time checking upon network updates. An extension of VeriFlow allows using assertions to check network properties during controller execution [119]. Similar to `SDNRacer`, these tools can detect interesting invariant violations. However, they cannot tell what precise sequence of events led to them, only that the latest update triggered the violation. STS [120] extends these works by considering the minimal sequence of events responsible for a given invariant violation. Unlike `SDNRacer`, STS does not have a precise formal specification of the partial orderings of events or the conditions under which two operations commute. As a result, STS cannot detect bugs unless the invariant is violated in a given trace. On the other hand, SDNRacer reports strictly more violations than STS by generalizing the observed trace to all traces obtainable from the same inputs. Additionally, STS uses network-wide snapshots to check various properties while SDNRacer considers all relevant events and thus does not miss any harmful violations. Finally, the output of SDNRacer and STS is different. STS outputs the minimal sequence of input events that reproduce an invariant violation while SDNRacer outputs the exact pairs of read/write events that caused the property violation.

Controller Verification. Other approaches seek to eliminate controller bugs, for instance, by synthesizing provably correct controllers [121, 122]. Similarly, in FlowLog [43], rulesets are partially compiled to NetCore [44] policies and then verified.

NICE [123] uses concolic execution of Python controller programs with symbolic packets and then runs a model checker to determine invariant violations. Kuai [124] uses a simplified version of an OpenFlow switch as well as a custom controller language, but applies partial order reduction techniques to reduce the number of states the model checker has to explore. Although significantly more performant, Kuai still suffers from the state-space explosion problem associated with full model checking. Vericon [125] converts programs into first-order logic formulas and uses a theorem prover to verify safety properties. In contrast, SDNRacer is a dynamic analyzer that operates on actual controller traces and can quickly detect concurrency issues: the root cause of many bugs. The speed of the analysis only depends on the trace size, not on the controller. Previous approaches could benefit from our formal specifications in order to speed-up their verification time; e.g., by not checking operations that do not interfere with the network state.

CONGUARD uses similar techniques as SDNRacer to detect and exploit concurrency violations inside the SDN controller software [126]. This work is orthogonal to SDNRacer since SDNRacer focuses on the concurrency violations that are induced by the read/write events from the switches, while CONGUARD focuses on the concurrency violations inside the SDN controller.

Moreover, researchers [127, 128] presented formal model-checking techniques to validate SDN controllers. While model checking techniques provide extended coverage of the application, the controller developers need to develop a faithful model of their implementation. The work of Jagadeesan *et al.* [129] overcomes some of the traditional scalability limitations of model-based verification by leveraging machine-learning algorithms.

Consistency properties. OpenFlow is asynchronous, and there is research on how to provide consistency guarantees under certain circumstances; e.g., how to ensure per-packet or per-flow consistency during policy changes. Reitblatt *et al.* introduced the notion of update consistency [130]. Several papers [130–133] introduced different formal techniques to preserve these properties during network updates. Researchers developed domain-specific languages such as the Frenetic language [14], the FatTire language [134], and NetCore [44] which aim to eradicate bugs in OpenFlow networks altogether. This is achieved by adding layers of abstraction and forcing the developers to use higher-level constructs or declarative languages instead. These are then executed by a run-time system or compiled by a compiler that ensures correctness. Developers using such systems can use more abstract thinking but must trust that the run-time system being used is sound. `SDNRacer` complements this work and can be used to verify that the run-time system is correct. The authors of Attendre [135] propose a mechanism using versioned flow-table entries and packet buffers that would eliminate certain classes of race conditions in OpenFlow networks. The main problem with these solutions is that they do not scale: as flow tables and buffers are always of finite size, solutions relying on them cannot make any guarantees for the worst case when the network is congested. However, they do help in situations where there is guaranteed to be enough space available for a given task; i.e., the flow table should have enough space to hold more than one policy version.

Troubleshooting tools. OFRewind [136] enables manual debugging of OpenFlow networks by running as a proxy between OpenFlow switches and controllers. It can capture both control and data plane traffic, and can re-inject traces into the network. Similar efforts are made with ndb [137], a network debugger that sits between the controller and switches and enables tracing of packets across the network. These tools can be useful for debugging, but they do not explore reorderings of events. In the case of ndb, additional barrier messages are inserted to prevent reorderings during debugging.

Violation Clustering and Ranking. Researchers applied grouping or clustering concurrency violation for event-driven concurrency analyzers in other domains [138–143]. The main difference is that our clustering method is based on fine-grained semantic happens-before information rather than coarse-grained indicators (e.g., whether an operation in a violation is in the framework [138]). Also, `SDNRacer` does not rely on static analysis and actually considers the controller code as a black box. Thanks to this, our clustering approach based on happens-before information is general and can thus benefit existing analyzers such as EventRacer for Android [138].

`SDNRacer` also goes beyond reducing the number of false positives produced by traditional concurrency analyzers by automatically reasoning about the common causes underlying the violations using domain-specific knowledge [102, 103, 144].

6 CONCURRENCY ANALYSIS FOR SOFTWARE-DEFINED NETWORKS

In this chapter, we present a system called `SDNRacer`, the first comprehensive dynamic and controller-agnostic concurrency analyzer for production-grade SDN controllers. `SDNRacer` checks for a variety of errors including (high-level) data races, packet coherence violations, and update isolation violations. It precisely captures the asynchrony of SDN environments; thanks to the first formulation of a *happens-before* (HB) model [104] for the most commonly used OpenFlow features. Our HB relation is based on an in-depth study of the OpenFlow specification [20] and the behavior of network switches [145]. Further, we present a *commutativity specification* of an SDN switch under which two operations on the switch commute. This specification elegantly abstracts the behaviors of the switch and is a principled approach to reducing the number of false positives, enabling precise and scalable analysis. To assist the SDN controller developers to debug and fix the reported concurrency violations, `SDNRacer` automatically identifies the most representative concurrency violations: those that capture the cause of the violation. The two key insights behind this assist are that: *(i)* many violations share the same root cause, and *(ii)* violations with the same cause share common characteristics. `SDNRacer` leverages these observations to cluster reported violations according to the similarity of events in them as well as SDN-specific features. `SDNRacer` then reports the most representative violation for each cluster using a ranking function.

We illustrate the practicality of `SDNRacer` by analyzing real-world SDN controllers (both, single and multi-threaded) and show that it automatically discovers harmful and previously unknown concurrency errors.

6.1 OVERVIEW

Now, we provide an overview of `SDNRacer`. We explain, in Section 6.1.1, how `SDNRacer` analyzes a trace of SDN network's events and identify harmful concurrency issues. In Section 6.1.2, we discuss how to use `SDNRacer` to detect violations of higher-level properties (beyond races) such as consistency violations. Finally, in Section 6.1.3, we discuss how `SDNRacer` aid developers to troubleshoot and debug concurrency issues.

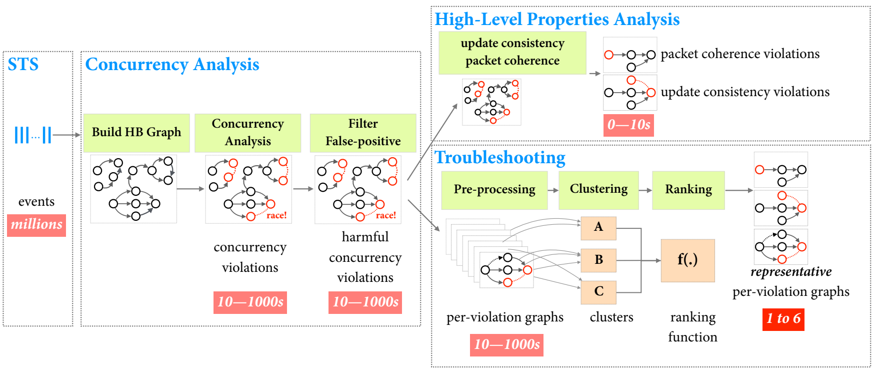


FIGURE 6.1: The pipeline of SDNRacer. SDNRacer accepts an event trace from production or emulated SDN network. First, SDNRacer analyzes the event trace identifying harmful consistency violations. Then, SDNRacer analyzes the reported races and identify if any causes a violation of high-level properties. To aid the developers, out of potentially thousands of violations, SDNRacer reduces them to a handful of representative ones which closely map to actual controller bugs.

6.1.1 *Concurrency Analysis*

Detecting concurrency issues, such as described in Section 5.1, requires a careful and precise definition of the operations in the network and of how ordering between operations is induced in the network. We begin by defining a small set of events which succinctly encapsulate the relevant operations performed by the controller, the network switches, and hosts in the network. The goal of defining these events is to design an abstraction that captures all the relevant information for the concurrency analysis without having to analyze irrelevant low-level implementation details (e.g., how a switch buffers a packet). Example of these events are PACKET_IN, PACKET_OUT, and FLOW_MOD presented in Section 5.1. SDNRacer accepts a trace of events produced by SDN network in production or emulated environment. We provide full specifications of the relevant events in Section 6.2.

Then, we define a precise happens-before (HB) model [104] for SDN networks. Happens-before is a binary relation that defines an ordering of events such that if event *a* happens before event *b* then the results of event *a* (e.g., modifying a flow table) are visible to event *b*. We provide the first happens-before specifications for SDN networks in Section 6.3. Events unordered by happens-before relation can interfere; e.g., events ⑦ and ⑨ in the example in Section 5.1. Not all interfering events are harmful to the network. For example, events that writes or reads different parts of the flow space (e.g., different IP addresses) do not need strict happens-before ordering; hence are considered false positive concurrency issues.

Filtering False-Positive Concurrency Issues

A key problem that every practical concurrency analyzer must address is reducing the amount of reported issues that are false positives and therefore, harmless. `SDNRacer` filters numerous false positives by leveraging two distinct filters. Together, these two filters reduce the number of races by up to 99.97%. Detailed evaluation of the filtering performance of our tool is provided in Section 6.8.

Filter 1: Commuting Events. Commutativity relates to whether changing the order of two events affects the network state in different ways. If not, then even if two events are interfering with each other (via low level reads and writes), the network state ends up being identical. Such interference is therefore harmless and can be filtered out.

Consider Figure 5.1 again and the write events ③–④ that are pushed to $S1$ upon the reception by the controller of the packet sent by Host #1 ②. These two write events race with each other as the switch does not guarantee any ordering between write requests: either ③ will happen before or ④. However, the race is harmless as the two events are for non-overlapping entries of the forwarding table. In other words, the forwarding table at $S1$ will end up being identical independently of whether ③ happens before ④ or not. We say that ③ and ④ *commute*. We present a precise formal definition of the commutativity specification of the forwarding table in Section 6.4.

Filter 2: Time. In theory, SDN switches can take an unbounded amount of time to perform a command (read or write). In practice, though, they tend to execute them within a relatively short time frame. This observation enables `SDNRacer` to filter unlikely interference issues [146–148]. For instance, if a read and a write event are separated by, say 10 seconds, then they are unlikely to be reordered in practice. `SDNRacer` enables the SDN developer to specify a time δ after which two events cannot interfere anymore. This δ can easily be estimated based on the maximum network delay and the maximum switch processing time.

6.1.2 *Detecting Violations of High-Level Properties*

`SDNRacer` goes beyond detecting interferences and is capable of detecting violations of higher-level properties such as inconsistent packet forwarding during a network update [130]. In particular, `SDNRacer` uses the detected harmful concurrency violations to identify if any violates update isolation and packet coherence properties. Informally, `SDNRacer` analyzes harmful violation between events that belongs to different updates to detect any violations of update isolation property. Similarly, `SDNRacer` analyzes harmful violation between packet read events and flow table write events to detect any violations of packet coherence property. We provide a formal specification of the high-level properties and how `SDNRacer` uses the results of the concurrency analysis to detect any violations of these properties in Section 6.5.

So far, only a few SDN controllers such as Frenetic [14] guarantee update consistency. With `SDNRacer`, an SDN developer can now analyze *any* controller for consistency problems. In Section 6.8, we show that many such controllers (Floodlight [149], POX [150], ONOS [15]) are actually inconsistent. Most importantly, `SDNRacer` consistency analysis enabled us to discover previously unknown harmful bugs in several of them.

6.1.3 *Assist with Debugging*

`SDNRacer` detects and reports all true-positive concurrency violations—every single one of them—which can amount to thousands even in minutes-long traces. Analyzing and troubleshooting all these violations is tedious for at least four reasons. *First*, using classical debugging tools that require replaying the log traces and fixing issues one-by-one is infeasible as the concurrency violations are often non-deterministic and hard to reproduce in test environments. *Second*, violations originating from the same bug might differ (either subtly or vastly), which makes them hard to classify manually. In our previous example, Section 5.1, violations would differ in the number of "bounces" observed between the controller and the switch. Worse, multiple switches can also be involved leading to a combinatorial explosion in the number of distinct violations. *Third*, the number of violations induced by each bug can vary significantly (Section 6.8) and does not necessarily correlate with the importance of the problem. *Fourth*, a developer has no information on the number of bugs that are causing the violations. A naive approach to randomly fix one violation at a time and then re-run the analysis can, therefore, take a long time to converge and be sub-optimal (especially, if done in a greedy way).

To aid the debugging process, `SDNRacer` aims to present the developer with only the representative violations which, ideally, correspond to the actual bugs. This allows SDN developers to focus on addressing the most serious cases. `SDNRacer` reduces the number of concurrency violations according to a three-step process; see Figure 6.1. Here we give a high-level overview of the three-steps process, and we describe in detail each step in Section 6.6.

Step 1: Pre-processing. Out of a given execution trace, a concurrency analyzer will typically build a directed graph according to a happen-before (HB) relation (where event a is connected to b if a happens before b). The analyzer will then report a concurrency violation for any two events which are unordered in the graph (i.e., are disconnected), both events access the same location, and one is a write event.

As `SDNRacer` needs to compare violations together, a pre-processing step first produces one sub-graph per violation given the HB-graph. This sub-graph only contains the events that led to the violation.

Step 2: Clustering. Given a set of per-violation graphs, `SDNRacer` clusters these graphs into a number of (ideally, representative of the bugs) classes. `SDNRacer` initializes the clustering process by grouping all isomorphic per-violation graphs. The intuition is that because these graphs share the same sequence and structure of events, they are more likely to exert the same code path (and therefore the same bug) inside the controller.

While isomorphic-based clustering is efficient at identifying "look-alike" violations, different violations from the same bug can take different shapes (e.g., different number of switches trigger the bug). Therefore, in the second phase, to reduce the number of clusters, `SDNRacer` applies a clustering strategy based on whether two per-violation graphs are similar to each other. `SDNRacer` defines this similarity based on a *distance* defined over a set of domain-specific features. If two per-violation graphs exhibit the same features, `SDNRacer` considers them similar to each other and clusters them together. `SDNRacer` uses several features for distance computation; for instance, two violations are closer to each other if both have a packet bouncing between the controller and the switch (as described in our example).

Step 3: Ranking. Since the number of clusters reported by `SDNRacer` is very low (6 or less in all experiments), each of the clusters contains many violations, sometimes on the order of 1,000s. In the final step, `SDNRacer` uses a ranking function to select "the most interesting" violation representative of the entire cluster. The ranking function identifies the most commonly occurring features in each cluster. Then, the ranking function selects the per-violation graphs that exhibit the most features and selects the smallest of these, thus, showing the simplest representative graph.

6.2 FORMAL MODEL OF SDN OPERATIONS

In this section, we define a formal model of a Software-Defined Network. This model includes both events occurring in the network as well as a model of the flow table in an OpenFlow switch. In later sections, we use this formalization to specify a precise happens-before (HB) relation and a commutativity specification of the flow table.

6.2.1 *Operations and Events*

We begin by defining a small set of events which succinctly encapsulate the relevant operations performed by the controller, the network switches, and hosts in the network. The operations are defined in Section 6.2.2 and contain the *reads* and *writes* (updates) to the flow table.

For each event type, we define a set of attributes that describe the event. Depending on the event type, only a subset of the attributes $\langle pid, mid, out\_pids, out\_mids, msg\_type, sw, ops\rangle$ is used. Where pid is the identifier of the packet processed by the event. Since network packets are potentially processed by more than one event, SDNRacer generates a Packet ID pid that does not map directly to any of the headers but rather it designates a specific packet in a specific event. mid is the identifier of the OpenFlow message processed by the event. If there are no such packets/messages, these attributes are set to the undefined value $\perp$. The set out_pids contains the identifiers of all packets emitted by the event. For each event that emits a packet (e.g., SendPkt), SDNRacer generates a new unique pid for the packet and add it to the event's out_pids set. Each out_pids is a set because events emitting multiple packets generates multiple new pids. For instance, SDN switches can duplicate packets and output them on multiple ports. The HB model uses the Packet IDs to link causally related events as defined in Section 6.3. The set out_mids contains the identifiers of all OpenFlow messages emitted by the event. Each out_mids is a set because the controller can issue multiple messages in response to one event. If there are no such packets or messages, these sets are empty $\varnothing$. For events where $mid \neq \perp$, the OpenFlow message processed by the event is of type msg_type. The relevant message types for our analysis are: PACKET_IN, PACKET_OUT, BARRIER_REQUEST, BARRIER_REPLY, PORT_MOD, FLOW_REMOVED and FLOW_MOD. Finally, sw is a switch identifier, and ops is the set of flow table operations the event contains.

The following events capture the behavior of the switches, controllers, and hosts:

HandlePkt(sw, pid, out_pids, out_mids, ops**)** denotes that a switch sw received and processed a data plane packet pid. There are three cases: *(i)* the switch generates OpenFlow messages. In this case, out_mids contains the OpenFlow messages and out_pids contains the packet stored in the switch buffer; *(ii)* the switch forwards the packet. In this case, out_pids contains the packet to be forwarded, or; *(iii)* the switch drops the packet.

HandleMsg(sw, mid, pid, out_pids, out_mids, msg_type, ops**)** denotes that a switch sw received and processed the OpenFlow message mid with type msg_type. The pid is $\perp$ unless the switch reads a packet from its buffer to process the OpenFlow message (e.g., PACKET_OUT OpenFlow message). As a result of processing this packet, the switch can generate OpenFlow messages, and in such case, out_mids contains the OpenFlow messages identifiers. Moreover, the switch can generate new packets; in which case out_pids contains new packets identifiers.

SendPkt(sw**,** pid**,** out_pids**)** denotes that a switch sw sent the packet pid with a new identifier (in out_pids) out to another switch or host.

SendMsg(sw**,** mid**,** out_mids**)** denotes that a switch sw sent the OpenFlow message mid out to the controller with the identifier in out_mids.

RemovedFlow(sw**,** mid**,** out_mids**,** ops**)** denotes that a flow table entry in the switch timed out or a message mid explicitly deleted it. As a result of this event, the switch might generate an OpenFlow flow removed message; in which case the out_mids contains its identifier.

CtrlHandleMsg(mid**,** out_mids**)** denotes that the controller received and processed the OpenFlow message mid, and generated the OpenFlow messages in out_mids in response. If the controller did not generate a response, then out_mids is empty.

CtrlSendMsg(mid**,** out_mids**)** denotes that the controller sent the OpenFlow message mid out to the control plane with the identifier in out_mids.

HostHandlePkt(pid**,** out_pids**)** denotes that a host received and processed the packet pid, and generated the packets in out_pids in response.

HostSendPkt(pid**,** out_pids**)** denotes that a host sent the packet pid with a new identifier (in out_pids) out to another switch or host.

6.2.2 *A Model of an SDN Flow Table*

We now define a model of the flow table in an OpenFlow switch which contains a set of entries used to match packets.

Flow Table: Entries

A packet contains a *header* and a *payload*. The *header* consists of a set of fields (e.g., IP source, IP destination, or VLAN ID). The switch uses the *header* fields to match packets against flow table entries. The *payload* is a sequence of bits and does not affect our specification. For a packet pkt, we use the notation $pkt.h$ to refer to the header associated with pkt.

Each flow table entry contains the fields *match*, *priority*, *actions*, *counters*, and *timeouts*. The *match* is a boolean predicate over a packet header's fields, and the boolean predicate can be either an exact match (i.e., matching a single value) or a wildcard match (i.e., using bit masks). *Priority* is a number specifying entry preference in case the packet matches multiple flow entries, and *actions* specify a set of forwarding operations the switch must perform on a matching packet. *Counters* contain values used for statistics, while *timeouts* contain hard and idle timeout values.

For a flow table entry e, we use the shortcut notation $e.m$, $e.p$ and $e.a$ to refer to the *match*, *priority*, and *actions*, respectively. A match between two entries e_1 and e_2 is exact, denoted as $e_1.m = e_2.m$, when all *match* fields are exactly the same (including the wildcards). A match between e_1 and e_2 is wildcard, denoted as $e_1.m \subseteq e_2.m$, if some of the fields in $e_1.m$ are not an exact match but contained in $e_2.m$ due to more permissive wildcards. The same definition of wildcard and exact match applies to a packet and to a flow table entry.

Flow Table: Operations

An OpenFlow switch performs four types of operations on its flow table: $read$, add, $mode$, and del. $read$ operations are part of $HandlePkt$ event. The switch performs a $read$ operation on the flow table for each packet it receives. While add, mod, and *(iv)* del operations are part of $HandleMsg$ events with a msg_type of FLOW_MOD and the switch performs these operations when an OpenFlow message is processed. We define the semantics of the flow table operations, according to OpenFlow specification 1.0 [20], as follow:

1. $read(pkt)/e_{read}$: A $read$ operation denotes that a switch matches a packet pkt against the flow table to determine the highest priority flow table entry e_{read} that should be applied. If there is no such flow table entry, then e_{read} is the empty value $none$. Note that the value of e_{read} depends on the state of the flow table when the switch matched the packet pkt.

2. $add(e_{add}, no\_overlap)$: An add operation denotes that a switch tries to add a new entry e_{add} to the flow table. When $no\_overlap$ is true, then the switch does not add the new entry if there is exists an entry in the flow table with the same priority and the match of both entries overlap (i.e., there exists at least one packet that may match both the new entry the existing entry).

3. $mod(e_{mod}, strict)$: A mod operation modifies existing entries in the flow table. A boolean flag $strict$ is used to distinguish between the two types of modifications issued by the controller. In strict mode, the switch uses an exact match (including the priorities) to determine whether it should modify an entry whereas in non-strict mode the switch uses a wildcard match. Note that the switch treats *mod* as an *add* in case no match is found.

4. $del(e_{del}, strict)$: A *del* operation deletes all entries that match the entry e_{del} in the flow table. Similarly to the *mod* operation, *strict* affects how the switch performs the match operation to determine which entries to remove.

6.3 HAPPENS-BEFORE MODEL

In this section, we define a precise happens-before (HB) model for SDN networks based on the events described in Section 6.2. To ensure the correctness of the happens-before model, we designed the model based on an in-depth study of the OpenFlow switch specification [20] and the analysis of two software switch implementations: the POX software switch as well as the production quality Open vSwitch [145].

The HB relation is a binary relation $\prec \subseteq Event \times Event$ that is irreflexive and transitive. For convenience, we use the notation $\alpha \prec \beta$ instead of $(\alpha, \beta) \in \prec$. For a finite trace consisting of a sequence of events $\pi = \alpha_0 \cdot \alpha_1 \cdot \dots \cdot \alpha_n$ we use $\alpha <_\pi \beta$ to denote that event α occurs before event β in π. We use *HandleMsgs* to denote a set of all the events of type *HandleMsg* and define such sets for each event type defined in Section 6.2. We illustrate the HB ordering rules induced from a given trace π in Figure 6.2. All except four rules (BarrierPre, BarrierPost, Time1, Time2) make use of the information provided by the attributes *pid*, *out_pids*, *mid*, and *out_pids*. These capture the causality between two events α and β in the trace, where α caused β to happen. BarrierPre, BarrierPost describe the effect of BARRIER_REQUEST messages on OpenFlow switches. The rules Time1 and Time2 are speculative (discuss later).

We next proceed to describe our rules. We also illustrate the effect of each rule on the example shown in Figure 5.1.

SwitchDataPlane: This rule orders events that process packets within a single switch. It orders events that generate a given packet before the *SendPkt* events sending the given packet to another switch. In the background example, this rule introduces the ordering ⑤ ≺ ⑥.

SwitchControlPlane: This rules orders events that process OpenFlow messages within a single switch. It orders events that generate a given OpenFlow message before the *SendMsg* events sending the given message to the controller. In the background example, this rule introduces the orderings ① ≺ ②, and ⑦ ≺ ⑧.

SwitchBuffer: When sending a PACKET_IN OpenFlow message to the controller in a *SendMsg* event, the full packet contents need not be contained inside the message. Instead, the switch may store the packet in its buffer and send only a part of the packet to the controller. Later, a *HandleMsg* event of *msg_type* PACKET_OUT

SwitchDataPlane:

$$\frac{\alpha \in \mathit{HandlePkts} \cup \mathit{HandleMsgs} \qquad \beta \in \mathit{SendPkts} \qquad \beta.\mathit{pid} \in \alpha.\mathit{out\_pids}}{\alpha \prec \beta}$$

SwitchControlPlane:

$$\frac{\alpha \in \mathit{HandlePkts} \cup \mathit{HandleMsgs} \cup \mathit{RemovedFlows} \qquad \beta \in \mathit{SendMsgs} \qquad \beta.\mathit{mid} \in \alpha.\mathit{out\_mids}}{\alpha \prec \beta}$$

SwitchBuffer:

$$\frac{\alpha \in \mathit{HandlePkts} \cup \mathit{HandleMsgs} \qquad \beta \in \mathit{HandleMsgs} \qquad \beta.\mathit{pid} \in \alpha.\mathit{out\_pids}}{\alpha \prec \beta}$$

Host:

$$\frac{\alpha \in \mathit{HostHandlePkts} \qquad \beta \in \mathit{HostSendPkts} \qquad \beta.\mathit{pid} \in \alpha.\mathit{out\_pids}}{\alpha \prec \beta}$$

Controller:

$$\frac{\alpha \in \mathit{CtrlHandleMsgs} \qquad \beta \in \mathit{CtrlSendMsgs} \qquad \beta.\mathit{mid} \in \alpha.\mathit{out\_mids}}{\alpha \prec \beta}$$

Dataplane:

$$\frac{\alpha \in \mathit{SendPkts} \cup \mathit{HostSendPkts} \qquad \beta \in \mathit{HandlePkts} \cup \mathit{HostHandlePkts} \qquad \beta.\mathit{pid} \in \alpha.\mathit{out\_pids}}{\alpha \prec \beta}$$

ControlplaneTo:

$$\frac{\alpha \in \mathit{SendMsgs} \qquad \beta \in \mathit{CtrlHandleMsgs} \qquad \beta.\mathit{mid} \in \alpha.\mathit{out\_mids}}{\alpha \prec \beta}$$

ControlplaneFrom:

$$\frac{\alpha \in \mathit{CtrlSendMsgs} \qquad \beta \in \mathit{HandleMsgs} \qquad \beta.\mathit{mid} \in \alpha.\mathit{out\_mids}}{\alpha \prec \beta}$$

BarrierPre:

$$\frac{\alpha, \beta \in \mathit{HandleMsgs} \qquad \alpha.\mathit{msg\_type} = \texttt{BARRIER\_REQUEST} \qquad \alpha.\mathit{sw} = \beta.\mathit{sw} \qquad \alpha <_\pi \beta}{\alpha \prec \beta}$$

BarrierPost:

$$\frac{\alpha, \beta \in \mathit{HandleMsgs} \qquad \beta.\mathit{msg\_type} = \texttt{BARRIER\_REQUEST} \qquad \alpha.\mathit{sw} = \beta.\mathit{sw} \qquad \alpha <_\pi \beta}{\alpha \prec \beta}$$

Time1:

$$\frac{\alpha \in \mathit{HandlePkts} \cup \mathit{HandleMsgs} \qquad \beta \in \mathit{HandleMsgs} \qquad \beta.t - \alpha.t > \delta}{\alpha \prec \beta}$$

Time2:

$$\frac{\alpha \in \mathit{HandleMsgs} \qquad \beta \in \mathit{HandlePkts} \cup \mathit{HandleMsgs} \qquad \beta.t - \alpha.t > \delta}{\alpha \prec \beta}$$

Figure 6.2: Happens-before rules capturing ordering of events in a trace π.

or FLOW_MOD may retrieve the packet from the buffer before processing it. This rule orders *HandlePkt* and *HandleMsg* events that store a packet in the switch buffer before the *HandleMsg* event that eventually retrieves a packet from the switch's buffer. In the example, this rule introduces the ordering ① ≺ ⑤.

Host: This rule orders the processing of the packet in a *HostHandlePkt* event before the sending of the reply packets in *HostSendPkt* events.

Controller: This rule orders the processing of the OpenFlow message in a *CtrlHandleMsg* event before the sending of the reply messages in *CtrlSendMsg* events. In the example, this rule introduces the orderings ② ≺ ③, ② ≺ ④, ② ≺ ⑤, ② ≺ ⑨, and ② ≺ ⑩.

DataPlane: This rule orders events that send a packet before events that receive the packet. In the example, this rule introduces the ordering ⑥ ≺ ⑦.

ControlPlaneTo and ControlPlaneFrom: These rules order events that send an OpenFlow message before events that receive the message. In our example, these rules order the send of ②, ③, ④, ⑤, ⑧, ⑨, and ⑩ before the respective receive.

Barrier: For performance reasons, the switch is allowed to handle messages received from the controller in a different order from the one they were sent. To enforce ordering, the controller can issue a BARRIER_REQUEST message which ensures that the network switch finishes processing of all previously received messages (enforced by BarrierPre rule), before executing any messages beyond the BARRIER_REQUEST (enforced by BarrierPost rule). Note that the switch sends BARRIER_REPLY message to the controller once it finished processing BARRIER_REQUEST and all the messages before it.

Speculative time-based rules This rule adds edges between events that are highly unlikely to be reordered due to the physical limits of the network. The value of δ depends on the specific parameters of the network. It should include the maximum delay that a packet might take traversing the network and the time window in which the OpenFlow switches can reorder write events. The proper value of δ can be inferred from related work that measured flow setup time in different environments and switches from various vendors [146–148]. We show the effect of choosing δ in Section 6.8.2.

6.4 COMMUTATIVITY SPECIFICATION

In this section we introduce a commutativity specification for an OpenFlow switch. This is an important component that has been used previously to improve concurrency of multicore systems [151] as well as to enhance the precision of program analyses dealing with interference [152]. Here, the commutativity specification is important to reduce the number of reported false positives. As with the HB model, we designed the model based on an in-depth study of the OpenFlow switch specification [20] and experimental testing with Open vSwitch [145].

To define what *commutativity* means, we compare the results of two operations, in particular, flow table state and the returned values (if any) of the participating operations. We consider two flow tables to be in the same state if all their flow table entries contain identical *priority*, *match*, and *actions* fields. For the purposes of commutativity we ignore the *counters* and *timeout* fields as they are not used for matching packets or entries.

The commutativity specification is conveniently specified in a form of a logical predicate φ over pairs of operations. For a pair of operations a and b, the predicate φ_b^a evaluates to $true$ if operations commute and to $false$ otherwise.

Auxiliary Relations We define three auxiliary functions. First, we overload the set intersection operator $e_1 \cap e_2$ for entry match structures $e.m$ (and packet headers $e.p$) and use it to compute all packet headers that may match both e_1 and e_2. Next, we use $e_1 \overset{strict}{\subseteq} e_2$ to model the semantics of table entry matching in regular and $strict$ modes as follows:

$$e_1 \overset{strict}{\subseteq} e_2 \quad := \quad \begin{array}{ll} e_1.m = e_2.m \wedge e_1.p = e_2.p & \text{if } strict \\ e_1.m \subseteq e_2.m & \text{if } \neg strict \end{array}$$

A $deletes$ predicate models the semantics of a delete operation and specifies whether an entry e can be deleted:

$$deletes(e_{del}, e, strict) := e \overset{strict}{\subseteq} e_{del} \wedge e.out\_port \subseteq e_{del}.out\_port$$

Commutativity Specification. We present the full commutativity specification of an OpenFlow switch in Figure 6.3. We write the rules in the form that specifies when the operations do not commute which is then negated. We adopt this approach as the resulting rules are more intuitive to read. What follows is a description of some of the non-trivial rules.

$$\varphi^{read(pkt)/e_{read}}_{add(e_{add}, no\_overlap)} := \begin{cases} \neg(e_{read} \neq none \wedge e_{read} = e_{add}) & \text{if } add <_\pi read \\ \neg(pkt.h \subseteq e_{add}.m \\ \quad \wedge (e_{read} = none \vee (e_{read}.p \leq e_{add}.p \wedge e_{read}.a \neq e_{add}.a))) & \text{if } read <_\pi add \end{cases}$$

$$\varphi^{read(pkt)/e_{read}}_{mod(e_{mod}, strict)} := \begin{cases} \neg(e_{read} \neq none \wedge e_{read} \overset{strict}{\subseteq} e_{mod} \wedge e_{read}.a \neq e_{mod}.a) & \text{if } mod <_\pi read \\ \neg(e_{read} \neq none \wedge pkt.h \subseteq e_{mod}.m \wedge e_{read}.a \neq e_{mod}.a) & \text{if } read <_\pi mod \end{cases}$$

$$\varphi^{read(pkt)/e_{read}}_{del(e_{del}, strict)} := \begin{cases} \neg(pkt.h \subseteq e_{del}.m) & \text{if } del <_\pi read \\ \neg(e_{read} \neq none \wedge deletes(e_{del}, e_{read}, strict)) & \text{if } read <_\pi del \end{cases}$$

$$\varphi^{del(e_{del}, strict_{del})}_{mod(e_{mod}, strict_{mod})} := \begin{cases} \neg(deletes(e_{del}, e_{mod}, true)) & \text{if } strict_{mod} \\ \neg(e_{del}.m \cap e_{mod}.m \neq \varnothing) & \text{otherwise} \end{cases}$$

$$\varphi^{add(e_{add}, no\_overlap)}_{del(e_{del}, strict)} := \neg(deletes(e_{del}, e_{add}, strict) \vee (no\_overlap \wedge e_{add} \cap e_{del} \neq \varnothing))$$

$$\varphi^{mod(e_1, strict_1)}_{mod(e_2, strict_2)} := \begin{cases} \neg(e_1.m \cap e_2.m \neq \varnothing \wedge e_1.a \neq e_2.a) & \text{if } \neg strict_1 \wedge \neg strict_2 \\ \neg(e_1.m = e_2.m \wedge e_1.p = e_2.p \wedge e_1.a \neq e_2.a) & \text{if } strict_1 \wedge strict_2 \\ \neg((e_1 \overset{strict_2}{\subseteq} e_2 \vee e_2 \overset{strict_1}{\subseteq} e_1) \wedge e_1.a \neq e_2.a) & \text{otherwise} \end{cases}$$

$$\varphi^{add(e_{add}, no\_overlap)}_{mod(e_{mod}, strict)} := \begin{cases} \neg(e_{add} \overset{strict}{\subseteq} e_{mod} \wedge e_{add}.a \neq e_{mod}.a) & \text{if } \neg no\_overlap \\ \neg(e_{add} \cap e_{mod} \neq \varnothing) & \text{otherwise} \end{cases}$$

$$\varphi^{add(e_1, no\_overlap_1)}_{add(e_2, no\_overlap_2)} := \begin{cases} \neg(e_1.m \cap e_2.m \neq \varnothing \wedge e_1.p = e_2.p) & \text{if } no\_overlap_1 \vee no\_overlap_2 \\ \neg(e_1.m = e_2.m \wedge e_1.p = e_2.p \wedge e_1.a \neq e_2.a) & \text{otherwise} \end{cases}$$

FIGURE 6.3: Commutativity specification of an OpenFlow switch. Note that two *read* or two *del* operations always commute.

$\varphi(add, add)$**:** Adding two entries does not commute if: *(i)* the second entry overwrites the first one, or *(ii)* the second entry is not added because the first entry is already in the table. The entries can overwrite each other only if both are added without the $no\_overlap$ option and their $match$ and $priority$ are identical. In this case, the old entry is replaced with the new one and as long as their actions are different they do not commute. If at least one entry specifies the $no\_overlap$ option, then they do not commute if they have the same $priority$ and there exists an entry that can be matched by both entries.

$\varphi(add, mod)$**:** In case the $no\_overlap$ option is not set, add and mod do not commute in cases when they are allowed to modify the same entry with different actions. If $no\_overlap$ is set, then mod can add a new entry that overlaps with add which would result in add not being added.

$\varphi(del, mod)$**:** If mod affects only a single entry ($strict$ mode), we simply check whether this entry can be deleted. Otherwise, as long as both rules can match the same entry, they do not commute.

$\varphi(add, del)$**:** add and del do not commute if: (i) the added entry can be removed by a subsequent delete, or (ii) the delete does not remove the entry to be added but might enable adding it by removing some other entries. This situation arises when headers that may match add and del overlap.

$\varphi(mod, mod)$**:** If neither modify operation uses $strict$ mode then they do not commute if there is an entry that may match both. If they are both $strict$ then this entry needs to be exactly the same. Otherwise they do not commute if they are allowed to change the entry of each other.

$\varphi(read, add/mod/del)$**:** For $read$ operations we distinguish two cases depending on the order in which the operations are executed in the trace. If a $read$ happens first, the operations do not commute if the matched entry is not guaranteed to match after the second operation is performed. Since we know the concrete flow entry that matched the initial read, such a check can be performed precisely. In the case of a $read$ executing second, we simply check whether the matched rule is identical to the one added or modified. For a del operation, we conservatively check whether an entry that matches the packet can be removed.

Key Points Note, that for the $read$ operations our commutativity specification incorporates parts of the flow table state by using the returned values. Further, commutativity rules for $read$ are specialized based on the trace order, which is a direct consequence of depending on the state in which the operations were performed. However, commutativity checking remains efficient, as no flow table state beyond these return values needs to be stored or simulated.

6.5 CONSISTENCY PROPERTIES

In this section, we discuss the checking of two important previously defined consistency related properties in Section 6.5.2 and Section 6.5.3. A useful guarantee of our checking approach is that if we establish the properties holding on a single trace, it follows that the properties hold for all traces which contain the same events (though perhaps events appear in a different order) where the traces use the same input (Section 6.5.4). This guarantee reduces the number of traces we need to explore per input.

6.5.1 *Network Update*

SDN applications typically update more than one flow rule in the network to reflect entire policy changes; e.g., re-routing congested traffic through a different end-to-end path. To capture this behavior, we map individual events containing write operations in a trace π into sets of network updates, such that each set Γ of network updates reflects a policy change in the network. Network updates are either triggered reactively by messages from switches (e.g., PACKET_IN messages), or proactively by an external event (e.g., manual change from a network operator).

In reactive applications such as a learning switch, we can use the happens-before model to extract the set Γ of events that are part of a reactive update for the event α. For reactive updates, α is of type $RemovedFlow$ or $SendMsg$. More formally, a reactive update for event α is the set of events defined as follows:

$$
\begin{aligned}
&R(\pi,\alpha) ::= \\
&\{\beta \mid \beta \in \pi \land \alpha \prec \gamma_1 \prec \gamma_2 \prec \beta \\
&\quad \land\ \gamma_1 \in CtrlHandleMsgs \quad \land\ \gamma_1 \in Succ(\alpha) \\
&\quad \land\ \gamma_2 \in CtrlSendMsgs \quad\ \ \land\ \gamma_2 \in Succ(\gamma_1) \\
&\quad \land\ \beta \in Succ(\gamma_2)\}
\end{aligned}
$$

where $Succ(\gamma)$ returns all events created directly as a result of processing event γ:

$$Succ(\gamma) ::= \{e \mid e.pid \in \gamma.pid\_outs\}$$

On the other hand, in proactive applications, such as a static flow pusher, the updates are caused by external events or internal controller configurations and hence are outside the scope of our HB model. We provide two options to group proactive write events into network updates. The controller can annotate the writes with the version number. Alternatively, to keep controller instrumentation to a minimum, we provide a heuristic to detect proactive updates. The heuristic uses a clustering algorithm to group events together based on time into a set Γ of network updates. Then, we merge different clusters if there is a barrier request in one cluster and the response in another. This merge operation mitigates clustering errors from slow network updates.

COMMUTATIVITY RACE Beyond standard read-write data races, a core high level property that we check is commutativity races [152]. A commutativity race occurs when two events: *(i)* do not commute according to our commutativity specification, and *(ii)* the events are unordered by our happens-before relation. Given a trace π, we denote the set of commutativity races in π as $CR(\pi)$.

Further, for the reported commutativity races the same guarantees as in existing state-of-the-art commutativity happens-before race detectors [152] are provided. In particular: *(i)* the first reported race is always guaranteed to be a real race, and *(ii)* if no race is reported for the given execution, then no execution from the same input state contains a race.

6.5.2 *Update Isolation*

Wang *et al.* [153] define a set of policy changes to be *isolated* if they do not interfere with each other. That is, executing the updates defined by each policy in any interleaving results in a network state that is equivalent to one that is obtained by some serial execution. We check if a set of multiple policy changes $\Gamma^\star = \{\Gamma_1, \Gamma_2, \Gamma_3, \dots\}$ is isolated, by checking if no pair of events (across different policy changes) is in the set of commutativity races $CR(\pi)$:

$$UI(\Gamma^\star) ::= \nexists\ \alpha, \beta : (\alpha, \beta) \in CR(\pi) \wedge \alpha \in \Gamma_u \wedge \beta \in \Gamma_v \wedge \Gamma_u \neq \Gamma_v$$

6.5.3 *Packet Coherence*

The next property we check is coherence of a packet trace. We say that a packet trace is coherent if each packet is processed entirely using one consistent global network configuration [130, 154]. To check for this property, given a trace π, we first define the notion of a packet trace which is a subset of events that participate in processing packet pkt as it traverses throughout the network until the packet reaches a destination host. An event trace $\tau(\pi, \gamma)$ is a subset of the events in trace π that were created as a result of processing event γ. We say that event trace $\tau(\pi, \gamma)$ corresponds to a packet trace for a given packet pkt if event γ originated the packet pkt. More formally, the event trace $\tau(\pi,\gamma)$ is defined as follows:

$$\tau(\pi,\gamma) ::= \gamma \cup \{\tau(\pi,\beta) \mid \beta \in Succ(\gamma) \wedge \beta \notin HostHandlePkts\}$$

Then, we write $CR_\tau(\pi,\gamma)$ to denote all races where one of the racing events is in $\tau(\pi,\gamma)$.

$$CR_\tau(\pi,\gamma) ::= \{(\alpha,\beta) \mid \alpha \in \tau(\pi,\gamma) \ \wedge \ (\alpha,\beta) \in CR(\pi)\}$$

We check packet coherence for all packets pkt in a given trace π, i.e., we check coherence for each packet trace $\tau(\pi,\gamma)$ extracted from the trace π. We can be certain that a packet trace $\tau(\pi,\gamma)$ exhibits packet coherence if $CR_\tau(\pi,\gamma) = \varnothing$: any network update that could affect the packet trace would introduce at least one race between the previous network state and the updated state.

However, under certain conditions a packet trace can be coherent in the presence of races, i.e., when $CR_\tau(\pi,\gamma) \neq \varnothing$. Then, there is packet coherence if *(i)* there is only a single event e (containing flow table read operations on a single switch sw) that is part of any races in $CR_\tau(\pi,\gamma)$, and *(ii)* there are no events in $CR_\tau(\pi,\gamma)$ that modify any switches other than sw.

$$\begin{aligned} PC(\pi,\gamma) ::= \quad & CR_\tau(\pi,\gamma) = \varnothing \\ & \vee(\exists\, e : (\forall(\alpha,\beta) \in CR_\tau(\pi,\gamma) : \alpha = e \\ & \wedge\forall(\alpha,\beta) \in CR_\tau(\pi,\gamma) : \beta.sw = e.sw)) \end{aligned}$$

Intuitively, this means that there can be packet coherence even in the presence of races, if the reordering of the single event in the races does not negatively affect packet coherence. This is possible if there is only a single such event, i.e., if there are only two possible reorderings.

6.5.4 *Guarantees*

We note that our checks for the properties discussed above are more general than simply taking snapshots of the flow tables [115, 117, 118], as verification of a static snapshot does not consider event reorderings. Even though a trace π may be free of violations, there may be another trace π' with the same inputs as π which does contain violations. In contrast, our checks on π guarantee that any such trace π' is free of violations, which is useful as it means we do not need to explore all possible traces π'. Our guarantee is standard in happens-before classic race detectors [102, 152], however, here we ensure the guarantee even beyond races.

6.6 ASSIST WITH DEBUGGING

As we observe in our evaluation, see Section 6.8, even the simplest bugs can trigger thousands of harmful concurrency violations due to the highly asynchronous nature of the network. Such a large number of reports can overwhelm the developers and make it hard to troubleshoot and debug the bug behind the concurrency violations.

In this section, we present how `SDNRacer` assists developers with troubleshooting and debugging SDN concurrency violations in three steps. In Section 6.6.1, we explain how `SDNRacer` pre-processes the happen-before graph to prepare it for the next two steps. In Section 6.6.2, we present the clustering algorithm that `SDNRacer` utilizes to group related violations into clusters. Finally, in Section 6.6.3., we describe how `SDNRacer` ranks the violations in each cluster to present the developer with the most representative violation in each cluster.

6.6.1 *Pre-processing*

`SDNRacer` starts by pre-processing the output of SDN concurrency analyzer: the directed graph induced by the happens-before relation (HB-graph) and a list of violations, to produce one graph per-violation with only the events that led to it.

We first show how `SDNRacer` reduces the size of the HB-graph. Then, we present how `SDNRacer` extracts sub-graphs to help analyzing each concurrency violation individually in later stages.

Trimming HB-Graph. While the number of events in each trace is large, not all of these events pertain to concurrency violations. Such events are filtered by `SDNRacer` to reduce the computational complexity of the following stages. Note, at this stage `SDNRacer` does not detect new violations and it does not remove any from the HB-graph.

`SDNRacer` removes three categories of events from the HB-graph. *First*, it removes all the events that occurred during the network initialization phase and did not cause any concurrency violation such as the handshake messages between each switch and the controller. *Second*, it removes all events that did not lead to a concurrency violation. *Third*, it removes any redundant HB edges in the HB-graph.

Extracting Per-Violation Graphs. Even after removing irrelevant events, the resulting HB-graph is still massive containing many events and concurrency violations. As we are interested in how individual concurrency violations compare with each other, `SDNRacer` isolates each one of them into a separate graph such that each graph contains a single concurrency violation with all the events that led to it. `SDNRacer` builds the violation graphs by performing an upward traversal of the

HB-graph starting from the two events involved in each violation until it reaches one of the entry points (e.g., host send or proactive update) present in the trace. Note that the violation graphs vary in size and a single event may appear in multiple graphs, if it causes more than violation.

6.6.2 *Hierarchical Clustering*

Now we describe `SDNRacer` hierarchical clustering process. We first show how `SDNRacer` relies on graph isomorphism to initialize the set of clusters. Next, we present a set of SDN-specific features that `SDNRacer` utilizes to measure how related two nonisomorphic violations are. Then, we present how `SDNRacer` uses the SDN-specific features to calculate a distance measure between different violations. Finally, we present that full clustering algorithm.

Cluster Initialization. `SDNRacer` first clusters each violation according to an isomorphic check, essentially grouping violations containing equivalent event sequences.

In `SDNRacer`, we restrict the notion of event equivalence to event type (not the actual content of the event). Specifically, we say that two violation graphs G and H are isomorphic (and therefore grouped in the same cluster) *if* each node in G can be exactly mapped to a node H with the same type and the same set of edges.

While checking for graph isomorphism can be done in quasipolynomial time [155], it can still take a long time to complete in practice. Therefore, `SDNRacer` uses a configurable timeout value (by default 10 sec) to limit the wall-clock time for the isomorphism computation. If `SDNRacer` could not complete the isomorphic test within the time budget, the two graphs are considered as not isomorphic and put in different clusters. Observe that they can still be clustered together in later stages. Note that a heuristic isomorphic test can be used instead to speed up `SDNRacer` cluster initialization (at the cost of accuracy) [156].

Identifying Related Violations Through SDN-Specific Features. As a second step, `SDNRacer` uses SDN domain-specific features computed over each graph to compute a distance matrix between clusters. This distance matrix is then used to refine the initial clustering, clustering together closely related (but not equivalent) violations. We identified these features by manually inspecting the similarities among violation graphs of many known bugs traces. Then, we tested the learned features against different known bugs in real controllers. Other features can be discovered using machine-learning techniques.

`SDNRacer` uses two different feature types: *(i) boolean features* that either exist or not in a violation graph, e.g., the graph has a packet flooding event; and *(ii) numerical* *features* that represent how often the feature is present in the graph, e.g., the number of HostSendPkt events.

Formally, let G_k be the set of graphs in cluster C_k, $F_i : G_k \to \mathbb{N}$ be a function that returns the number computed for feature i. If feature i is boolean, F_i returns 1 if a graph has this feature, 0 otherwise. If feature i is numerical, F_i returns the actual number of features.

We now present the seven different features currently implemented in `SDNRacer` (adding additional ones is easy).

1. **Controller/Switch bouncing:** This boolean feature captures repeated PACKET_IN and PACKET_OUT events between the controller and a given switch for the same given packet. This situation occurs when the controller does not use proper synchronization primitives to ensure the rule that matches the packet has been committed to the Flow Table before sending the PACKET_OUT back to the switch.

2. **Reply packets:** This boolean feature captures if host replying to a packet it received triggered the violation. Often, the controller simultaneously installs a bidirectional path for a flow. The intuition behind this feature is to consider concurrency violations affecting the same flow closer to each other.

3. **Flow expiry:** OpenFlow allows flow entries to expire after a certain specified (hard or soft) timeout [20]. While the soft timeout helps cleaning the flow table, defining the timeout is usually tricky in asynchronous environments. Often, early flow expiry leads to many concurrency violations. This boolean feature captures violations caused by a flow expiry event.

4. **Flooding:** Often controllers flood packets for various reasons; i.e., the controller discovering the network topology or it is not aware of the location of the destination host of a given packet. However, the paths and the event-ordering that follows a packet flood is completely non-deterministic (hence, not isomorphic). If miss-handled, flooded packets cause concurrency violations. The corresponding graphs are often entirely different. As such, this boolean feature simply captures if packet flooding caused the violation.

5. **Number of root events:** This feature returns the number of *root events* in the violation graph. A root event is an event with only outgoing edges in the violation graph; e.g., the HostSendPkt event for the first packet in a flow. The number of root events indicates if one or more events cause the violation.

6. **Number of host sends:** This feature returns the number of the host send events.

7. **Number of proactive violations:** SDNRacer distinguishes two types of events: *reactive* and *proactive* events. Reactive events are events sent by the controller in response to received messages (e.g., responding to PACKET_IN event), while proactive are sent independently (e.g., proactive network update). This feature returns the number of proactive events involved in the violation.

Our experiments and manual analysis of various HB-graphs indicated that not all features carry the same significance in relating two violations; see Section 6.8.4. For instance, violations sharing the *flooding* feature tend to be more related than violations sharing *reply packets* one. Next, we show how SDNRacer captures this effect in a distance function by assigning different weights to each feature.

Distance Calculation. After SDNRacer extracts the features of each graph in a given cluster, it computes the mean of each feature in the cluster. Let $\{g_1, \ldots, g_n\} \in G_k$ be the set of graphs in cluster C_k. The mean of feature i is computed as:

$$m_i^k = \frac{\sum_{l=1}^{l=|G_k|} F_i(g_l)}{|G_k|}$$

Our distance calculation algorithm then computes the distance between every two clusters per-feature. The computation treats boolean and numerical features differently. For boolean features, two clusters are closer to each other if they contain a similar number of occurrences of the feature. For numerical features, two clusters are closer to each other if they share the same mean. Specifically, SDNRacer computes the per-feature distance between two clusters C_l and C_k as:

$$d_i = abs(m_i^l - m_i^k) \qquad \begin{cases} 0 & \text{if } m_i^l = m_i^k \\ 1 & \text{if } m_i^l \neq m_i^k \end{cases}$$

if i is a boolean feature   if i is a numerical feature

We assign different weights for each of the j features. The distance between two clusters C_l and C_k is:

$$d = \sum_{i=1}^{j} w_i d_i$$

SDNRacer computes the distance matrix between all the clusters and then feed it to the clustering algorithm.

Then, the hierarchical clustering algorithm groups several clusters into new clusters. For the distance between these groups, SDNRacer uses the distance between the farthest neighbors (also known as *complete linkage*) as the distance between the clusters.

Clustering Algorithm. We now describe `SDNRacer`'s hierarchical clustering algorithm, which is a case of agglomerative clustering [157].

The six major steps of the algorithm are: *Step 1*, initialize the clusters using the isomorphic check. *Step 2*, evaluate all the pair-wise distances. *Step 3*, construct a distance matrix using distances values. *Step 4*, merge the cluster pairs with shortest distances and remove them from the distance matrix. *Step 5*, evaluate all distances from this new cluster to all other clusters and update the matrix. *Step 6*, repeat until the distance matrix is reduced to a single element or the distances are longer than a predefined threshold.

6.6.3 *Ranking*

While the clustering algorithm groups the concurrency violations into a small number of clusters, the number of violations per cluster is large, potentially in the order of 1,000s. `SDNRacer`'s ranking function selects the most representative violation for each cluster.

The primary intuition is to find the smallest graph that exhibits the most common features across all graphs in one cluster. The ranking function starts with examining the boolean features first. It selects all violation graphs that have all the boolean features exhibited in 50% or more of the reported violations in the cluster. The second stage is to reduce the set of chosen graphs based on the pre-computed numerical features. For numerical features, we chose the graphs with the minimum difference between the feature in the given graph and the overall mean for the cluster. The order of selecting the graphs based on the numerical features is: proactive violation events, the number of HostSendPkt events and finally the number of root events. For the final set of chosen graphs, our ranking function selects the graph with the minimum number of events to present to the controller developers.

6.7 IMPLEMENTATION

We implemented a full prototype of `SDNRacer` in around 3,000 lines of Python code[1]. The implementation consists of three parts: *(i)* an instrumentation of the SDN troubleshooting system STS [120]; *(ii)* an instrumentation of several controller frameworks (POX, Floodlight, ONOS); and *(iii)* a concurrency analyzer that implements the happens-before rules, commutativity checks, high-level properties

1 `https://github.com/nsg-ethz/SDNRacer`

Controller	POX	FloodLight	ONOS
LoC	40	139	55

TABLE 6.1: While SDNRacer *does not* require controller instrumentation, adding few lines of instrumentation code enables to filter more harmless issues (around 20% more).

checks, and presents the developer with handful of representative concurrency violations.

Network Instrumentation. STS simulates a entire network, including OpenFlow switches, links, and hosts [120]. We instrumented STS to further track packets, messages, and switch operations and write them to a file. We remark that SDNRacer is not dependent on STS and use any production-grade quality-assurance framework to obtain event traces similar to the ones produced by STS.

(Optional) Controller Instrumentation. The controller instrumentation for POX, Floodlight, and ONOS includes a wrapper around the respective event handlers for incoming messages, and links the incoming message with the corresponding outgoing message, when possible. Instrumenting the controller only requires a few lines of code (Table 6.1). The controller instrumentation then passes this information to STS. Instrumenting the controller is *not needed* for SDNRacer to work, but it helps in filtering harmless concurrency issues by adding more HB orderings in addition to those defined in Section 6.3 (e.g., from 314 to 239 reported races, 23.9%, in one experiment). POX uses cooperative threading and runs only one task at any given time while Floodlight and ONOS are multi-threaded and they context-switch threads. However, this is not relevant to our model because SDNRacer treats the controller as a blackbox, allowing us to use SDNRacer on a wide set of controllers with minimal instrumentation in the controller framework. A more specific approach would allow for more precision at the price of being controller-specific.

SDNRacer SDNRacer reads events from a trace file, builds the HB graph and then runs the concurrency analysis on top of it. The HB graph as well as the races and inconsistent packets are output graphically for further inspection.

6.8 EVALUATION

In this section, we evaluate SDNRacer's performance and usability. After describing our setup, in Section 6.8.1, we first show in Section 6.8.2 that SDNRacer detects many consistency issues in existing controllers. As the number of issues is often large, we also show that `SDNRacer` can efficiently reduce the number of reported issues through filtering. Second, we show in Section 6.8.3 several examples of consistency violations discovered by `SDNRacer`. Third, we show in Section 6.8.4 how `SDNRacer` is effective in selecting the most representative violations to assist the developers in fixing the underlying bugs that are causing the majority of the harmful violations. Finally, we show in Section 6.8.5 that `SDNRacer` is fast and completes its analysis in few seconds on large traces containing thousands of events. Our results indicate that `SDNRacer` is an effective tool for troubleshooting real-world SDN deployments.

6.8.1 *Experimental Setup*

We ran `SDNRacer` on a set of network traces collected from a representative set of SDN controllers, running different existing applications, on different network topologies.

SDN Controllers. We run `SDNRacer` against three controllers: Floodlight version 0.91 [149], POX EEL [150], and ONOS version 1.2.2 [15]. We further instrumented them to better track HB relation (Table 6.1).

SDN Applications. We choose 5 representative applications, including purely proactive and pure reactive applications. Unless specified otherwise, we run the same application on each controller. The implementation of all analyzed applications is included as part of the official controller distribution.

App#1. MAC-learning: A purely reactive application builds and maintains a dynamic MAC address table for each switch. This table maps known MAC addresses to the physical port on which they can be reached. We analyze the implementations shipped with Floodlight and POX [105, 106].

App#2. Forwarding: MAC-learning applications are highly inefficient as they work on a per-switch basis. To alleviate this, most controllers include a "Forwarding Application" which works at the network-level and reactively builds and maintains one network-wide MAC address table. We analyze the implementations shipped with Floodlight, POX, ONOS [107–110].

App#3. Circuit Pusher: This purely proactive application automatically installs paths between two hosts identified by their MAC addresses, as well as the switch and port they are connected to. We analyze the implementation shipped with Floodlight [111].

App#4. Admission Control: This SDN application allows/drops host communication based on given operator policies. We analyze the implementation shipped with Floodlight [112].

App#5. Load Balancer: This application performs stateless load balancing among a set of replica identified by a virtual IP address (VIP). Upon receiving packets destined to a VIP, the application selects a particular host and installs flow rules along the entire path. We analyze the implementation shipped with Floodlight [113].

Network Topologies. We ran each controller on three different topologies: *Single*, *Linear*, and *BinTree*. *Single* has one switch with two hosts. *Linear* has two switches with one host connected to each. *BinTree* has seven switches connected as a binary tree with four hosts connected to leaf switches.

Event Traces. We collected 29 traces using STS [120] and a mix of applications, controllers, and network topologies. The traces have between 193 and 24, 612 events spanning between 26 and 74 seconds (Table 6.2). Each trace is the result of 200 STS simulation steps. In every step, each host in the topology decides randomly whether it is going send a packet to another randomly chosen host.

Distance Function. A simple sensitivity analysis led us to use the following weights for the distance function (Section 6.6.2): The weight used for Controller/Switch Bouncing, Packet Flood, and Flow Expiry is 2. The weight used for the number of Proactive Violations is 1.5. While the weight used for the number of Host Sends is 1. The number of root events and Reply Packets has a weight of 0.5. The maximum distance for the merging of clusters was set to 2. We are aware that different weights can result in an even better (or worse) clustering and leave a full sensitivity analysis for later work.

Application Specific Parameters. Some applications required additional parameters to run. For Circuit Pusher, we install a new circuit every second between two randomly selected hosts as well as remove one existing circuit with a probability of 0.5. For Admission Control, we allow 80% of the hosts (randomly selected) to communicate. For Load Balancer, we create replica pools with two hosts and assign them a VIP. All hosts send traffic to the VIP. Since Load Balancer only makes sense with more than two hosts, we run it on larger topologies: (*Single4* and *Linear4*), connecting four hosts instead of two.

6.8.2 *Race Detection and Filtering Efficiency*

`SDNRacer` reports many races (Table 6.2) whose actual number depends on the number of read and write events which in turn depend on the controller running the application. As an illustration, the same set of inputs led to 16 reads and 66 writes for the MAC-learning application running on POX EEL, but only six reads and 66 writes when running on Floodlight.

			Events			Races				Updates		Packet Coherence		
App	Topology	Controller	Events	WR	RD	Races	Comm.	Time	Remain.	Num	¬ Isolt.	Pkts	Racing	Incoh
Learning	Single	POX EEL	193	7	42	294	218	66	**10 (3.40%)**	6	0	42	10	**0**
		Floodlight	314	7	70	494	223	227	**44 (8.91%)**	5	0	70	18	**0**
	Linear	POX EEL	274	16	66	532	387	121	**24 (4.51%)**	18	0	34	11	**5**
		Floodlight	233	6	66	190	64	125	**1 (0.53%)**	5	0	33	1	**0**
	BinTree	POX EEL	4033	487	663	62066	61337	664	**65 (0.10%)**	402	0	190	28	**18**
		Floodlight	9320	1251	904	275257	270217	4737	**302 (0.11%)**	1156	34	223	119	**72**
Forwarding	Single	POX Angler	106	4	16	61	33	21	**7 (11.48%)**	12	0	16	7	**0**
		POX EEL	145	8	19	109	84	24	**1 (0.92%)**	12	0	17	1	**0**
		POX EEL Fx	184	8	29	189	145	42	**1 (0.53%)**	12	0	26	2	**1**
		ONOS	476	18	71	1336	1163	127	**14 (1.05%)**	3	0	73	27	**22**
		Floodlight	97	3	13	35	13	14	**8 (22.86%)**	5	0	13	8	**0**
	Linear	POX Angler	248	13	48	323	116	191	**12 (3.72%)**	31	1	20	6	**6**
		POX EEL	306	20	50	405	235	159	**7 (1.73%)**	28	1	20	7	**7**
		POX EEL Fx	303	16	51	276	206	62	**4 (1.45%)**	27	0	20	3	**3**
		ONOS	880	44	181	4059	3781	228	**49 (1.21%)**	11	0	76	19	**15**
		Floodlight	180	6	36	104	46	45	**13 (12.50%)**	5	0	14	5	**5**
	BinTree	POX Angler	2106	286	359	20447	13179	6988	**272 (1.33%)**	127	4	77	43	**42**
		POX EEL	4362	504	453	34385	27956	6201	**219 (0.64%)**	138	3	86	59	**58**
		POX EEL Fx	4283	467	413	12509	12238	242	**24 (0.19%)**	147	0	92	61	**55**
		ONOS	8031	1492	920	236429	233578	2598	**239 (0.10%)**	37	0	131	66	**49**
		Floodlight	1886	203	323	12293	11766	317	**209 (1.70%)**	71	0	76	57	**53**
CircuitPusher	Single	Floodlight	218	25	41	1301	1040	218	**43 (3.31%)**	8	1	41	35	**0**
	Linear	Floodlight	327	42	74	1933	1581	287	**65 (3.36%)**	10	1	38	34	**21**
	BinTree	Floodlight	1200	144	227	6156	5605	507	**44 (0.71%)**	14	3	142	10	**6**
Adm. Ctrl.	Single	Floodlight	190	3	36	104	35	62	**6 (5.77%)**	5	0	36	7	**0**
	Linear	Floodlight	221	6	48	139	56	69	**14 (10.07%)**	6	0	21	6	**6**
	BinTree	Floodlight	841	52	170	1384	1090	228	**66 (4.77%)**	25	0	74	20	**10**
LoadBalancer	Single4	Floodlight	3889	822	476	703864	685158	16492	**2214 (0.31%)**	449	1114	77	22	**0**
	BinTree	Floodlight	24612	6213	2163	4705379	4642118	62031	**1230 (0.03%)**	1419	464	226	101	**97**

TABLE 6.2: Reported races and properties violations for different traces with applying time filter using $\delta = 2$. The numbers in bold are the final numbers of races and incoherent packets reported to the user.

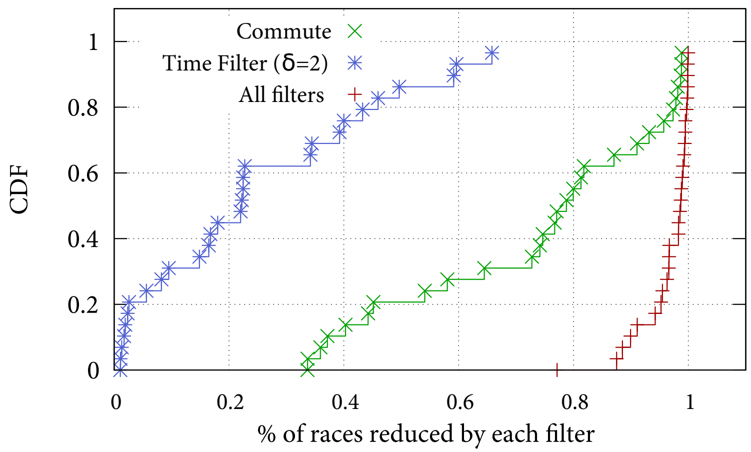


FIGURE 6.4: *The effect of `SDNRacer` filters.* When all filters are applied, more than 90% of all races are filtered in 89% of the cases. Commutativity filtering is the most efficient, followed by Time-based filtering.

Reporting too many races is of little use to the developer. So, to be of practical use, `SDNRacer` is equipped with a set of filters based on commuting events, timing, and race coverage [158]. We now evaluate the efficiency of each filter in turn. When all filters are applied, `SDNRacer` manages to filter out more than 90% of the races in the vast majority of cases.

Filter 1. Commutativity. Commutativity is a major contributor to reducing the number of reported races. This filter alone reduces at least 33% of the races in almost all traces and more than 73% of races in 65.5% of the traces (Figure 6.4).

Commutativity filtering performs best in traces that have many unrelated reads and writes. This high number of disjoint reads and writes is often the result of different hosts sharing the same path. For example, 91% of all races reported by running Circuit Pusher on the BinTree topology commute (Table 6.2) as the events are related to different hosts and non-overlapping entries.

Filter 2. Time-Based. Time filtering further helps reduce more than 20% of the races in about half of our traces (Figure 6.5) with a δ value of 2 seconds; see Section 6.3.

In Figure 6.5, we report filtering as a function of δ. If we set δ to a high value, `SDNRacer` reports more false-positive races. For instance, when we set δ to 8 seconds, the time filter can only reduce up to 34.5% of the races in its best case. In contrast, `SDNRacer` can filter up to 51.7% of the races in its best case when we set δ to 2 seconds.

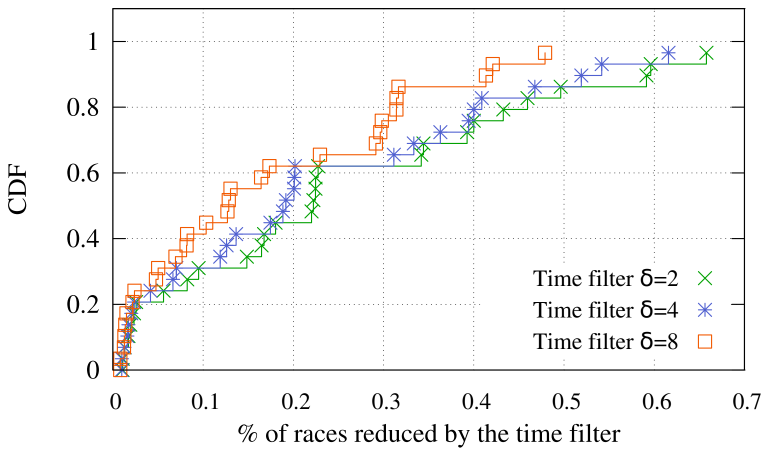


FIGURE 6.5: *The effect of time-based filter.* Choosing smaller δ filters more races. With δ=2, `SDNRacer` can filter more than 20% of the races in 48% of the cases. While with δ=8, `SDNRacer` can filter 20% of the races in 40% of the cases.

For our evaluation, 2 seconds is safe; given our switch implementation and network size.

Other Filters. Like all happens-before detectors (e.g., FastTrack [102]), `SDNRacer`'s checks are as precise as the happens-before model. Hence, there can be false positives for covered races [158] due to data dependencies. To discover such cases, in addition to commutativity-based and time-based filtering, `SDNRacer` provides an additional filter that discovers covered races. Covered races are reported interferences that cannot happen because of high-level dependencies. We observed, however, that covered races account for only up to 2.4% of the races. As such, to speed-up processing, `SDNRacer` does not enable that filter by default. Actually, we discovered (Figure 6.6) that covered races are redundant with time-based filtering. If time-based filtering is not applied, then 10% of races in 50% of the cases can be filtered.

6.8.3 *Consistency Checks*

`SDNRacer` detected consistency violations in all applications and controllers used in our experiments. In many cases, these violations turned out to be subtle (and some are unknown) bugs. In the following, we detail both update isolation and packet coherence violations (Section 6.5). Recall that the former leads to ultimately different

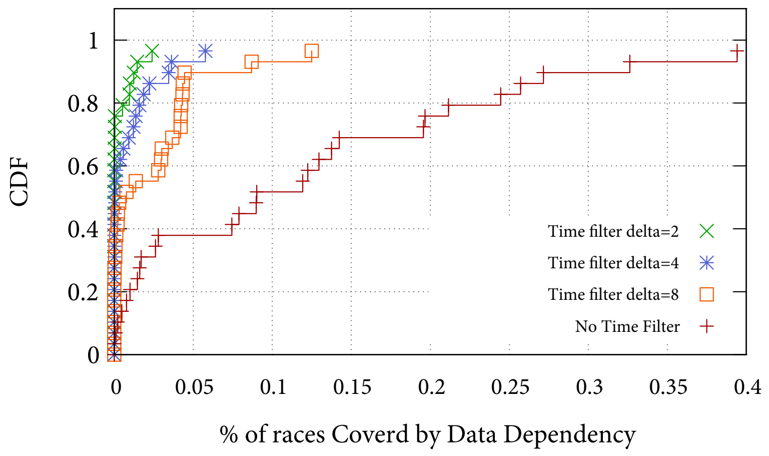


FIGURE 6.6: *The effect of covered races.* The covered race can filter more than 10% of the races in 50% of the cases by itself. However, its effect is minimal when used after the time filter.

network states being installed in the network while the latter relates to packets being forwarded according to different policies.

Violations of Update Isolation. `SDNRacer` discovered update consistency violations in four applications (in particular, 10 out of 29 traces): MAC-Learning, Forwarding, Circuit Pusher, and Load Balancer. For the Load Balancer application, the violation was the source of a severe bug.

- *Violation#1: Floodlight Load Balancer distributes flows inconsistently.* `SDNRacer` reports 1114 inconsistent network updates on the single switch topology and 464 inconsistent network update on the BinTree topology (Table 6.2).

  At first glance, the number of update isolation violations might seem high. However, the vast majority of the violations are symptoms of the same bug. By analyzing the reported violations by `SDNRacer` and the application code we realized that, upon packet reception, the controller selects a replica, pushes flow rules to direct traffic to it, sends the packet back in the network towards the replica *without* waiting for the flow rule to be committed to the switch. As the controller installing the flow table entries, further packets go to the controller and trigger the replica selection process again. Concretely, this means that the controller is taking multiple load-balancing decisions for the same flow. Inconsistent flow assignments can lead to bad performance but also leads to connection drops as the install rules to send packets from the same flow can to different replicas.

Fix: The bug is easily fixed by forcing the Load-Balancer to request a barrier before pushing packets back into the network and by having it buffer (or drop) any subsequent packets it receives for the same connection.

- *Violation#2: POX forwarding module deletes rules installed by other modules.* `SDNRacer` reports an inconsistent update where the removal of a flow induced by one module raced with a flow insertion induced by another module. Investigating the application code, we found out that the rules installed by the Discovery module (in charge of learning the network topology) were deleted by the Forwarding module whenever the topology changed.

  In this specific case, the race between the two modules is not harmful as the default OpenFlow 1.0 action is to direct packets to the controller. This ensured that even though rules from the Discovery modules were deleted, it was still able to learn the topology. We stress that in newer versions of OpenFlow, the default action is now to drop packets, meaning this bug would cause the entire network traffic to be dropped whenever the topology changes.

  Fix: This bug is easily fixed by ensuring that the Forwarding application only deletes its own flow rules.

Violations of Packet Coherence. `SDNRacer` discovered per-packet coherence violations in almost all traces (Table 6.2). Most of the incoherent packet cases concerned races occurring when the controller installs a set of flow rules and then sends a packet matching these flow rules without waiting for the flow rules to be committed first. As such, these type of races occurred more often in traces of reactive applications such as the Forwarding application. While waiting for writes to be committed is an obvious solution, it also slows down network operations, indicating that many controllers trade consistency for speed. In general, violating per-packet coherence may not always be harmful. Poorly performed policy updates, for instance, can create per-packet coherence violations without leading to data losses. Even in this case, we believe that it is still important to report and quantify violations of per-packet coherence as correctness predicated on policy content is undesirable.

6.8.4 *Selecting Representative Violations*

`SDNRacer` reports a small number of concurrency violations to the developer, moreover, fixing only these reported violations significantly reduces the number of violations exhibited by the controller.

		Concurrency Analyzer		Clustering		Cluster Sizes	
App	Controller	Events	Violations	Isomorphic	# Final Clusters	Median	Max
Adm. Ctrl.	Floodlight	908	81	26 (32.10 %)	**3 (3.70 %)**	24	33
CircuitPusher	Floodlight	1017	39	6 (15.38 %)	**2 (5.13 %)**	19.5	32
Forwarding	Floodlight	3016	288	58 (20.14 %)	**3 (1.04 %)**	31	215
	POX EEL	5632	310	160 (51.61 %)	**4 (1.29 %)**	64.5	143
LearningSwitch	Floodlight	6658	344	210 (61.05 %)	**5 (1.45 %)**	48	155
	POX EEL	3408	66	61 (92.42 %)	**2 (3.03 %)**	33	46
LoadBalancer	Floodlight	17593	1910	272 (14.24 %)	**5 (0.26 %)**	204	1362

TABLE 6.3: SDNRacer clustering performance on traces computed over a binary tree topology (200 steps, median on 15 repetitions).

Table 6.3 shows that SDNRacer is able to reduce the number of reported violations by up to three orders of magnitude. Clustering isomorphic graphs already reduces the number of violations by more than 66% in 50% of all cases. The feature-based agglomerative clustering further reduces them to less than 95% of the reported violations of SDNRacer in 50% of the cases.

Usability. To demonstrate the usability of SDNRacer, we used the reported violations by SDNRacer to fix the Floodlight Load Balancer application. When tested on the fixed version, the number of clusters reported by SDNRacer dropped from 3 to 2 and the number of violations was reduced by 99.23%.

It is worth mentioning that not all concurrency violations in SDN networks are fixable. This due to the inherent lack of OpenFlow synchronization primitives that order packets upon entering or while traversing the network. However, recent SDN systems give the controller the ability to synchronize packet entry to the network [159].

6.8.5 *Time*

SDNRacer finishes its concurrency and property violations analysis in less than 32 seconds in the vast majority of traces (Figure 6.7). To measure this, we ran SDNRacer 20 times and collected the total time for: *(i)* loading the trace; *(ii)* building the HB graph; *(iii)* applying all filters; and *(iv)* performing all consistency analysis.

The worst case (3.7 minutes) happened when SDNRacer analyzed the FloodLight Load Balancer on the BinTree topology. This long running time is due to a bug in the

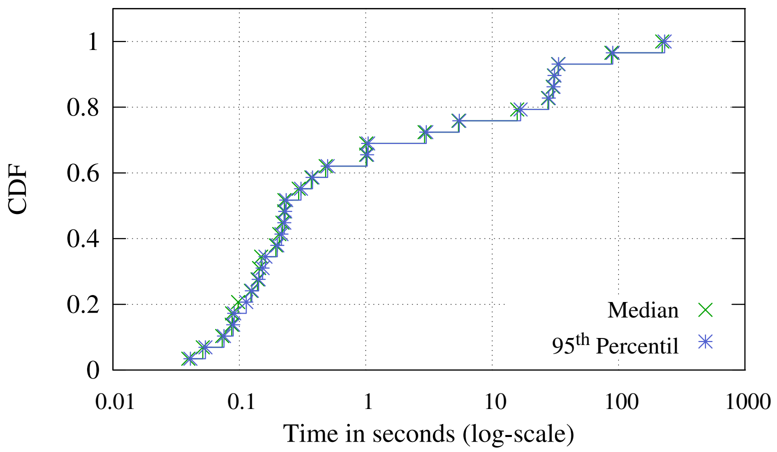


FIGURE 6.7: Analysis time for traces from Table 6.2. In 90% of the cases SDNRacer can analyze the traces in less than 30 seconds.

application (see Section 6.8.3) that caused the trace to have an order of magnitude more events and races than other traces.

Clustering Time. For the experiments reported in Table 6.3, SDNRacer finished in less than one second for 60 % of all the experiments, and only 5% of the experiments took more than 216 seconds with a worst case of 21 minutes (for Floodlight Load Balancer on binary tree).

The 10 seconds timeout for isomorphic check was triggered in only 2.08% of the checks of POX EEL Forwarding Application on binary tree. This application has a flooding-related concurrency violation which creates large violation graphs spanning the entire network.

SDNRacer can process subsets of the trace (windowing) which would be helpful to troubleshoot longer traces. The intuition here is that individual races are usually "concentrated" and do not last over the entire trace.

6.9 SUMMARY

In this chapter, we presented SDNRacer, the first scalable analysis system for finding a variety of concurrency-induced errors including (high-level) data races, per-packet consistency, and update consistency. SDNRacer makes several key contributions: *(i)* a precise formal happens-before model of SDN (OpenFlow) concurrency; *(ii)* efficient filters including a commutativity specification of a network switch; *(iii)* clustering and ranking algorithms aim to help developers to troubleshot and debug the underlying bugs in the controller; and *(iv)* a thorough experimental evaluation illustrating that our techniques for filtering races and identifying high-level (consistency) violations work in practice. `SDNRacer` was also able to identify previously unknown and harmful bugs in existing SDN controllers.

7 CONCLUSIONS AND OPEN PROBLEMS

In this dissertation, we developed techniques and tools that increase the network's reliability by automating the configuration process of networks running distributed routing protocols and verifying that SDN networks are free from concurrency errors.

We demonstrated with a practical example in Chapter 2 that human-induced misconfigurations could cause policy violations and network downtimes. We then proposed using network-wide configurations synthesis to avoid human-induced misconfigurations and listed the key challenges to build a practical network-wide configurations synthesis framework. Therefore, in this dissertation, we developed two frameworks to synthesize network-wide configurations: `SyNET` and `NetComplete`.

Our first solution posed the network-wide configurations synthesis problem as an instance of synthesizing inputs to stratified Datalog programs; presented in Chapter 3. This insight allowed our system, `SyNET`, to be general and to synthesize configurations for any protocol that is specified as a stratified Datalog program. Then, we developed a novel algorithm to synthesize inputs for stratified Datalog programs; a problem that has not been solved in previous work. We evaluated `SyNET` and showed that it could be used to synthesize configurations for networks with less than 64 routers. However, the generality of `SyNET` prevented it from scaling to larger networks and generating configurations that are similar to those written by humans.

In Chapter 4, we revisited the network-wide configurations problem and built a new system, `NetComplete`, with a focus on scalability and practicality. In `NetComplete`, we developed more efficient synthesis techniques optimized for each of the commonly used routing protocols. Moreover, we used configurations sketches to reduce the search space of possible configurations and to produce configurations closer to human-written ones. We demonstrated in our comprehensive set of experimental results that `NetComplete` can autocomplete configurations for large networks with up to 200 routers within a few minutes.

In Chapter 5, we introduced the notion of concurrency errors in SDN networks and demonstrated how such errors could affect production networks; causing policy violations and network downtimes.

In Chapter 6, we presented `SDNRacer`, the first scalable analysis system for finding a variety of concurrency-induced errors; including data races, per-packet consistency, and update consistency. To find concurrency-induced errors, we developed a precise model to capture the asynchronous of SDN networks. We demonstrated in our thorough experimental evaluation that SDNRacer effectively filters races and identifying high-level (consistency) violations work in practice. SDNRacer was also able to identify previously unknown and harmful bugs in existing SDN controllers.

7.1 OPEN PROBLEMS

Model Verification. Recent network synthesis and verification projects rely on a formal model of the network. For instance, NetComplete and Minesweeper [73] rely on a first-order logic model of the network, while SyNET and Batfish [67] rely on a Datalog model of the network. Thus, the correctness of these tools is dependent on the model being faithful to the network implementation. Often, the developers of such tools manually write the formal model that their tools use to synthesize or verify network configurations. To avoid formal modeling of the network, Plankton-neo [160] uses model checking techniques directly against the software implementation to verify middleboxes, but it is very dependent on the deterministic state restoration ability for the system under test. The question remains how we can derive correct abstract models of the network implementation. One approach is to learn the models directly from the network. Similar techniques have been employed before to learn program specifications [161–163]. However, in the context of networks, there are more challenges in terms of scalability and heterogeneity. We addressed the scalability challenges earlier in this thesis. For heterogeneity, there are multiple implementations of the same protocols from multiple vendors. A specification learning tool must understand what is common and different between these implementations. Another approach is to verify the correctness of the hand-written models against network implementation. In the second approach, data-plane verification tools [115, 117, 118, 125] can be used to verify that the computed forwarding state in the network matches the state that the model anticipated.

Synthesis for Quantitative Properties. While the existing network configurations synthesis frameworks focus on functional requirements (e.g., forwarding paths), synthesizing configurations that comply with quantitative properties remains an open problem. Quantitative properties include, for instance, producing configurations that give optimal convergence time, or a configuration that has a minimal number of changes compared to the existing configuration or configurations that induce minimal complexity. A solution to this problem needs to employ different techniques. For instance, minimizing the changes in a configuration or the complexity of a configuration can be framed as a MAX-SAT problem, while the properties for a fast convergence configuration can be learned from a large set of examples. Combining the various techniques used to synthesize functional and quantitative requirements in one scalable framework increases the challenges of this problem.

Learning-Based Synthesis. Existing network configurations synthesis frameworks either generate configurations that do not resemble what a human operator would write (e.g., `SyNET` and Propane [37]) or require inputs from the operator to generate the desired configurations (e.g., sketches in `NetComplete` or bounds on the routing domains in Zeppelin [56]). To foster the deployment of synthesis tools, a synthesis framework need to generate configurations that are similar to what a human operator would write. One particular sub-problem would be to figure out how to automatically learn “good” sketches from existing configurations; e.g., sketches that capture best configuration practices or local configuration style.

Learning-Based Concurrency Analysis. While `SDNRacer` is a complete and sound concurrency analyzer, its output is very verbose and could overwhelm the developers. To assist the developers in troubleshooting the root cause of these errors, we developed a method to cluster related concurrency violations based on a fixed set of features. However, we discovered these features by manually examining traces of known concurrency bugs. Manually finding the set of features to correlate violations and the weights to use in the clustering algorithm is a tedious task and does not guarantee that we discovered all the related features. One interesting problem here is applying Machine Learning techniques to learn the set of features from traces of known bugs.

BIBLIOGRAPHY

1. Jenni Ryall. *Facebook, Tinder, Instagram suffer widespread issues* `http://mashable.com/2015/01/27/facebook-tinder-instagram-issues/` (cited on pages ix, 13).
2. Tom Strickx. *How Verizon and a BGP Optimizer Knocked Large Parts of the Internet Offline Today* `https://blog.cloudflare.com/how-verizon-and-a-bgp-optimizer-knocked-large-parts-of-the-internet-offline-today/` (cited on pages ix, 13).
3. Juniper Networks. *What's Behind Network Downtime? Proactive Steps to Reduce Human Error and Improve Availability of Networks* technical report (Juniper Networks, 2008) (cited on pages ix, 13).
4. Google. *Google Compute Engine Incident 16004* `https://status.cloud.google.com/incident/compute/16004` (cited on pages ix, 13).
5. Time Warner Cable. *Outage's Presss Release* `http://www.twcableuntangled.com/2014/08/twc-identifies-cause-of-internet-outage/` (cited on page ix).
6. *United Airlines jets grounded by computer router glitch* `http://www.bbc.com/news/technology-33449693` (cited on pages ix, 13).
7. *Google routing blunder sent Japan's Internet dark on Friday* `https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/`. 2017 (cited on pages ix, 13).
8. *Stock trading closed on NYSE after glitch caused major outage* `https://www.theguardian.com/business/live/2015/jul/08/new-york-stock-exchange-wall-street` (cited on pages ix, 13).
9. *BGPmon. Internet Prefixes Monitoring.* `http://www.bgpmon.net/blog/` (cited on page ix).
10. Algosec. *The State of Automation in Security* technical report (Algosec, 2016) (cited on pages ix, 13).
11. *Stay Up to Stay in Business – The Cost and Cause of Network Downtime* `https://www.packetdesign.com/blog/cost-and-cause-of-network-downtime/`. Accessed: 2019-05-04. 2016 (cited on pages ix, 13).

12. Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker & Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review* **38**, 69 (2008) (cited on pages x, 8).

13. Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford & David Walker. Modular SDN Programming with Pyretic. *Technical Reprot of USENIX* (2013) (cited on pages x, 9).

14. Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story & David Walker. *Frenetic: A Network Programming Language* in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming ICFP '11* (ACM, Tokyo, Japan, 2011) (cited on pages x, 9, 23, 56, 58, 85, 90).

15. Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, *et al. ONOS: Towards an Open, Distributed SDN OS* in *Proceedings of the 3rd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking HotSDN '14* (ACM, Chicago, Il, USA, 2014) (cited on pages x, 90, 110).

16. Carolyn Jane Anderson, Nate Foster, Dexter Kozen & David Walker. *NetKAT: Semantic Foundations for Networks* in *Proceedings of the 41st ACM SIGPLAN Symposium on Principles of Programming Languages POPL '14* (ACM, San Diego, CA, USA, 2014) (cited on pages x, 9, 23).

17. Jedidiah McClurg, Hossein Hojjat, Nate Foster & Pavol Černy. *Event-driven Network Programming* in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '16* (ACM, Santa Barbara, CA, USA, 2016) (cited on pages x, 9).

18. Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown & Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review* **38**, 105 (2008) (cited on page x).

19. Jan Medved, Robert Varga, Anton Tkacik & Ken Gray. *OpenDaylight: Towards a Model-Driven SDN Controller Architecture* in *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), IEEE 15th International Symposium on* (IEEE, 2014) (cited on page x).

20. *OpenFlow Switch Specification. Version 1.0.0* `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf` (cited on pages xi, 87, 94, 95, 98, 106).
21. Jon Postel. *RFC 791: Internet Protocol* 1981 (cited on page 3).
22. Steve Deering & Robert Hinden. *RFC 8200: Internet Protocol, Version 6 (IPv6) Specification* 2017 (cited on page 3).
23. Richard Colella, R Callon, E Gardner & Y Rekhter. *RFC 1629: Guidelines for OSI NSAP Allocation in the Internet* 1994 (cited on page 5).
24. Richard Bellman. On a Routing Problem. *Quarterly of applied mathematics* **16**, 87 (1958) (cited on page 6).
25. Lester R Ford Jr. *Network Flow Theory* technical report (Rand Corp Santa Monica Ca, 1956) (cited on page 6).
26. Edward F Moore. *The Shortest Path Through a Maze* (Bell Telephone System., 1959) (cited on page 6).
27. C Hedrick. *RFC 1058: Routing Information Protocol* 19855 (cited on page 6).
28. Donnie Savage, J Ng, S Moore, D Slice, Peter Paluch & R White. *RFC 7868: Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP)* 2016 (cited on page 6).
29. Charles E Perkins & Pravin Bhagwat. *Highly Dynamic Destination-Sequenced Distance-Vector routing (DSDV) for Mobile Computers* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '94* (ACM, London, United Kingdom, 1994) (cited on page 6).
30. J Chroboczek. *RFC 6126: The Babel Routing Protocol* 2011 (cited on page 6).
31. Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik* **1**, 269 (1959) (cited on page 6).
32. J Moy. *RFC 1583: OSPF Version 2* 1994 (cited on pages 6, 67).
33. Rob Coltun, Dennis Ferguson, John Moy & A Lindem. *RFC 2622: OSPF for IPv6* 2008 (cited on pages 6, 67).
34. R. W. Callon. *RFC 1195: Use of OSI IS-IS for Routing in TCP/IP and Dual Environments* 1990 (cited on pages 6, 67).
35. Yakov Rekhter, Tony Li & Susan Hares. *RFC 4271: A Border Gateway Protocol 4 (BGP-4)* 2006 (cited on pages 7, 8).
36. Matthew Caesar & Jennifer Rexford. BGP Routing Policies in ISP Networks. *IEEE Network* **19** (2005) (cited on page 9).

37. Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye & David Walker. *Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '16* (ACM, Florianopolis, Brazil, 2016) (cited on pages 9, 23, 24, 53, 56, 58, 123).

38. Kausik Subramanian, Loris D'Antoni & Aditya Akella. *Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks* in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages POPL '17* (ACM, Paris, France, 2017) (cited on pages 9, 23–25, 56, 58).

39. Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma & Ying Zhang. *PGA: Using Graphs to Express and Automatically Reconcile Network Policies* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '15* (ACM, London, United Kingdom, 2015) (cited on pages 9, 23).

40. Yifei Yuan, Rajeev Alur & Boon Thau Loo. *NetEgg: Programming Network Policies by Examples* in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks HotNets '14* (2014) (cited on pages 9, 25).

41. Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford & Paul Hudak. *Maple: Simplifying SDN Programming Using Algorithmic Policies* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '13* (ACM, Hong Kong, China, 2013) (cited on pages 9, 23).

42. Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer & Nate Foster. *Merlin: A Language for Provisioning Network Resources* in *Proceedings of the 10th ACM Conference on Emerging Networking Experiments and Technologies CoNEXT '14* (ACM, Sydney, Australia, 2014) (cited on pages 9, 23).

43. Tim Nelson, Andrew D Ferguson, Michael JG Scheer & Shriram Krishnamurthi. *Tierless Programming and Reasoning for Software-Defined Networks* in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI '14, Seattle, WA, USA* (USENIX Association, 2014) (cited on pages 9, 23, 84).

44. Christopher Monsanto, Nate Foster, Rob Harrison & David Walker. *A Compiler and Run-time System for Network Programming Languages* in *Proceedings of the 39th ACM SIGPLAN Symposium on Principles of Programming Languages POPL '12* (ACM, Philadelphia, PA, USA, 2012) (cited on pages 9, 84, 85).

45. *Open Network Operating System (ONOS) Intent Framework* `https://wiki.onosproject.org/display/ONOS/The+Intent+Framework` (cited on pages 10, 23).

46. Martin J Levy. *The deep-dive into how Verizon and a BGP Optimizer Knocked Large Parts of the Internet Offline Monday* `https://blog.cloudflare.com/the-deep-dive-into-how-verizon-and-a-bgp-optimizer-knocked-large-parts-of-the-internet-offline-monday/` (cited on page 13).

47. Tom Paseka. *How a Nigerian ISP Accidentally Knocked Google Offline* `https://blog.cloudflare.com/how-a-nigerian-isp-knocked-google-offline/` (cited on page 13).

48. D. Awduche, A. Chiu, A. Elwalid, I. Widjaja & X. Xiao. *RFC 3272: Overview and Principles of Internet Traffic Engineering* 2002 (cited on page 14).

49. Bernard Fortz, Jennifer Rexford & Mikkel Thorup. Traffic Engineering with Traditional IP Routing Protocols. *IEEE Communications Magazine* (2002) (cited on page 14).

50. Rüdiger Birkner, Dana Drachlser-Cohen, Laurent Vanbever & Martin Vechev. *Net2Text: Query-Guided Summarization of Network Forwarding Behaviors* in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI '18, Renton, WA, USA* (USENIX Association, 2018) (cited on page 14).

51. Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford & David Walker. *Composing Software Defined Networks* in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI '13, Lombard, IL, USA* (USENIX Association, 2013) (cited on page 23).

52. Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva & Laure Thompson. *A Coalgebraic Decision Procedure for NetKAT* in *Proceedings of the 42nd ACM SIGPLAN Symposium on Principles of Programming Languages POPL '15* (ACM, Mumbai, India, 2015) (cited on page 23).

53. *OpenDayLight (ODL) Group-Based Policy* `https://wiki.opendaylight.org/view/Group_Policy:Main` (cited on page 23).

54. Sanjai Narain, Gary Levin, Sharad Malik & Vikram Kaul. Declarative Infrastructure Configuration Synthesis and Debugging. *Journal of Network and Systems Management* **16**, 235 (2008) (cited on page 23).

55. Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitu Padhye & David Walker. *Network Configuration Synthesis with Abstract Topologies* in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '17* (ACM, Barcelona, Spain, 2017) (cited on pages 23, 24, 53, 58).

56. Kausik Subramanian, Loris D'Antoni & Aditya Akella. *Synthesis of Fault-Tolerant Distributed Router Configurations* in *Proceedings of the ACM on Measurement and Analysis of Computing Systems SIGMETRICS '18* (ACM, Irvine, CA, USA, 2018) (cited on pages 23, 24, 123).

57. Alexander J.T. Gurney, Xianglong Han, Yang Li & Boon Thau Loo. *Route Shepherd: Stability Hints for the Control Plane* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '12* (ACM, Helsinki, Finland, 2012) (cited on pages 23, 24).

58. Alexander J., T. Gurney, Anduo Wang Limin Jia & Boon Thau Loo. *Partial Specification of Routing Configurations* in *Proceedings of the 1st Workshop on Rigorous Protocol Engineering WRIPE '11* (IEEE, Vancouver, BC Canada, 2011) (cited on pages 23, 24, 26).

59. Eric Schkufza, Rahul Sharma & Alex Aiken. *Stochastic Superoptimization* in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS XVIII* (ACM, Houston, TX, USA, 2013) (cited on page 24).

60. Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan & Hongqiang Harry Liu. *Automatically Repairing Network Control Planes Using an Abstract Representation* in *Proceedings of the 26th Symposium on Operating Systems Principles SOSP '17* (ACM, Shanghai, China, 2017) (cited on page 25).

61. Yifei Yuan, Dong Lin, Rajeev Alur & Boon Thau Loo. *Scenario-based Programming for SDN Policies* in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies CoNEXT '15* (ACM, eidelberg, Germany, 2015) (cited on page 25).

62. Shambwaditya Saha, Santhosh Prabhu & 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* (ACM, Santa Clara, CA, USA, 2015) (cited on page 25).

63. Ehab S Al-Shaer & Hazem H Hamed. *Discovery of Policy Anomalies in Distributed Firewalls* in *Proceedings of IEEE INFOCOM 2004* (IEEE, Toronto, Canada, 2004) (cited on page 25).

64. Nick Feamster & Hari Balakrishnan. *Detecting BGP Configuration Faults with Static Analysis* in *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation, NSDI '05, Boston, MA, USA* (USENIX Association, 2005) (cited on page 25).

65. Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler & Shriram Krishnamurthi. *The Margrave Tool for Firewall Analysis* in *LISA* (2010) (cited on page 25).

66. Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su & P. Mohapatra. *FIREMAN: a toolkit for firewall modeling and analysis* in *S&P* (2006) (cited on page 25).

67. Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan & Todd Millstein. *A General Approach to Network Configuration Analysis* in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI '15, Oakland, CA, USA* (USENIX Association, 2015) (cited on pages 25, 26, 29, 33, 122).

68. Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella & Ratul Mahajan. *Fast Control Plane Analysis Using an Abstract Representation* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '16* (ACM, Florianopolis, Brazil, 2016) (cited on pages 25, 26).

69. Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy & Zachary 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'16* (ACM, Amsterdam, Netherlands, 2016) (cited on pages 25, 26).

70. Santhosh Prabhu, Ali Kheradmand, Brighten Godfrey & Matthew Caesar. *Predicting Network Futures with Plankton* in *Proceedings of the First Asia-Pacific Workshop on Networking* (ACM, New York, NY, USA) (cited on page 25).

71. Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman & George Varghese. *Checking Beliefs in Dynamic Networks* in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI '15, Oakland, CA, USA* (USENIX Association, 2015) (cited on page 25).

72. A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov & C. Talcott. FSR: Formal Analysis and Implementation Toolkit for Safe Interdomain Routing. *IEEE/ACM Transactions on Networking* **20**, 1814 (2012) (cited on pages 25, 26).

73. Ryan Beckett, Aarti Gupta, Ratul Mahajan & David Walker. *A General Approach to Network Configuration Verification* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '17* (ACM, Los Angeles, CA, USA, 2017) (cited on pages 26, 122).

74. Yannis Smaragdakis & Martin Bravenboer. *Using Datalog for Fast and Easy Program Analysis* in *Proceedings of International Datalog 2.0 Workshop* (Springer, Oxford, United Kingdom, 2010) (cited on page 26).

75. Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik & Hongseok Yang. *On Abstraction Refinement for Program Analyses in Datalog* in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '14* (ACM, Edinburgh, UK, 2014) (cited on pages 26, 27).

76. Magnus Madsen, Ming-Ho Yee & Ondřej Lhoták. *From Datalog to Flix: A Declarative Language for Fixed Points on Lattices* in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '16* (ACM, Santa Barbara, CA, USA, 2016) (cited on page 26).

77. Kryštof Hoder, Nikolaj Bjørner & Leonardo De Moura. *μZ: An Efficient Engine for Fixed Points with Constraints* in *Proceedings of the 23th International Conference on Computer Aided Verification CAV '11* (Springer, Snowbird, UT, USA, 2011) (cited on page 27).

78. Ethan K. Jackson & Janos Sztipanovits. *Towards a Formal Foundation for Domain Specific Modeling Languages* in *Proceedings of the 6th ACM International Conference on Embedded Software EMSOFT '06* (ACM, Seoul, Republic of Korea, 2006) (cited on page 27).

79. Ethan K. Jackson & Wolfram Schulte. *Model Generation for Horn Logic with Stratified Negation* in *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems FORTE'08* (Springer, Tokyo, Japan, 2008) (cited on page 27).

80. Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert & Thomas Santen. *Components, Platforms and Possibilities: Towards Generic Automation for MDA* in *Proceedings of the 10th ACM International Conference on Embedded Software EMSOFT '10* (ACM, Scottsdale, AZ, USA, 2010) (cited on page 27).

81. Cristian Cadar & Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM* (2013) (cited on page 27).

82. Daniel Kroening & Michael Tautschnig. *CBMC – C Bounded Model Checker* in *Proceedings of 20th International conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS '14* (Springer, Grenoble, France, 2014) (cited on page 27).

83. Edmund Clarke, Daniel Kroening & Flavio Lerda. *A Tool for Checking ANSI-C Programs* in *Proceedings of 10th International conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS '04* (Springer, Barcelona, Spain, 2004) (cited on page 27).

84. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia & Vijay Saraswat. *Combinatorial Sketching for Finite Programs* in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS XII* (ACM, San Jose, CA, US, 2006) (cited on pages 27, 67, 69).

85. Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe & Ion Stoica. *Declarative Networking: Language, Execution and Optimization* in *SIGMOD* (2006) (cited on pages 29, 33).

86. Inderpal Singh Mumick & Oded Shmueli. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence* (1995) (cited on page 30).

87. Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv & Oded Shmueli. Static Analysis in Datalog Extensions. *Journal of the ACM* (2001) (cited on pages 36, 44).

88. *Foundations of Databases: The Logical Level* (Eds. Serge Abiteboul, Richard Hull & Victor Vianu) (Addison-Wesley Longman Publishing Co., Inc., 1995) (cited on pages 37, 44).

89. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems* (Computer Science Press, 1989) (cited on page 38).

90. `https://logicblox.com/content/docs4/corereference/html/index.html`, (cited on page 48).

91. C. Barrett et al. *The SMT-LIB Standard: Version 2.0* 2010 (cited on pages 48, 70).

92. L. De Moura & N. Bjørner. *Z3: An Efficient SMT Solver* in *Proceedings of 14th International conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS '08* (Springer, Budapest, Hungary, 2008) (cited on pages 48, 54, 70).

93. *Graphical Network Simulator-3 (GNS3)* `https://www.gns3.com/` (cited on pages 49, 71).

94. Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Alistair Bowden & Matthew Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* (2011) (cited on pages 51, 71).

95. Theophilus Benson, Aditya Akella & David A. Maltz. *Mining Policies from Enterprise Network Configuration* in *IMC* (2009) (cited on page 51).

96. Jeff Doyle & Jennifer Carroll. *Routing TCP/IP, Volume 1* (Cisco Press, 2005) (cited on page 51).

97. Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang & Ahsan Arefin. *A Network-State Management Service* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '15* (ACM, London, United Kingdom, 2015) (cited on page 55).

98. Nanxi Kang, Ori Rottenstreich, Sanjay G Rao & Jennifer Rexford. Alpaca: Compact Network Policies With Attribute-Encoded Addresses. *IEEE/ACM Transactions on Networking* (2017) (cited on page 55).

99. G Gonzalo *et al. Network Mergers and Migrations: Junos Design and Implementation* (John Wiley & Sons, 2011) (cited on page 55).

100. Leslie G Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing* **8**, 410 (1979) (cited on page 68).

101. Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever & Martin Vechev. *Network-Wide Configuration Synthesis* in *Proceedings of the 29th International Conference on Computer Aided Verification CAV '17* (Springer, Heidelberg, Germany, 2017) (cited on pages 74, 75).

102. Cormac Flanagan & Stephen N. Freund. *FastTrack: Efficient and Precise Dynamic Race Detection* in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '09* (ACM, Dublin, Ireland, 2009) (cited on pages 79, 85, 103, 114).

103. Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards & Brad Calder. *Automatically Classifying Benign and Harmful Data Races Using Replay Analysis* in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '07* (ACM, San Diego, CA, USA, 2007) (cited on pages 79, 85).

104. Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* (1978) (cited on pages 82, 87, 88).

105. James McCauley. *POX EEL L2 Learning Switch* `https://github.com/noxrepo/pox/blob/eel/pox/forwarding/l2_learning.py`. 2015 (cited on pages 83, 110).

106. Big Switch Networks, Inc. *Floodlight Learning Switch* `https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/learningswitch`. 2013 (cited on pages 83, 110).

107. Big Switch Networks, Inc. *Floodlight Forwarding Application* `https://github.com/floodlight/floodlight/blob/v0.91/src/main/java/net/floodlightcontroller/forwarding/Forwarding.java`. 2013 (cited on pages 83, 110).

108. James McCauley. *POX Angler Forwarding Application* `https://github.com/noxrepo/pox/blob/angler/pox/forwarding/l2_multi.py`. 2012 (cited on pages 83, 110).

109. James McCauley. *POX EEL Forwarding Application* `https://github.com/noxrepo/pox/blob/eel/pox/forwarding/l2_multi.py`. 2015 (cited on pages 83, 110).

110. Open Networking Laboratory. *ONOS (Open Network Operating System): Forwarding Application* `https://github.com/opennetworkinglab/onos/tree/onos-1.2/apps/fwd`. 2015 (cited on pages 83, 110).

111. Big Switch Networks, Inc. *Floodlight Circuit Pusher Application* `https://github.com/floodlight/floodlight/tree/v0.91/apps/circuitpusher`. 2013 (cited on pages 83, 110).

112. Big Switch Networks, Inc. *Floodlight Firewall* `https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/firewall`. 2013 (cited on pages 83, 110).

113. Big Switch Networks, Inc. *Floodlight Load-Balancer Application* `https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/loadbalancer`. 2013 (cited on pages 83, 111).

114. Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey & Samuel Talmadge King. *Debugging the Data Plane with Anteater* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '11* (ACM, Toronto, ON, Canada, 2011) (cited on page 83).

115. Peyman Kazemian, George Varghese & Nick McKeown. *Header Space Analysis: Static Checking for Networks* in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI '12, San Jose, CA, USA* (USENIX Association, 2012) (cited on pages 83, 103, 122).

116. Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown & Amin Vahdat. *Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks* in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI '14, Seattle, WA, USA* (USENIX Association, 2014) (cited on page 83).

117. Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar & P Godfrey. *Veriflow: Verifying network-wide invariants in real time* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '12* (ACM, Helsinki, Finland, 2012) (cited on pages 83, 103, 122).

118. Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown & Scott Whyte. *Real Time Network Policy Checking Using Header Space Analysis* in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI '13, Lombard, IL, USA* (USENIX Association, 2013) (cited on pages 83, 103, 122).

119. Ryan Beckett, Xuan Kelvin Zou, Shuyuan Zhang, Sharad Malik, Jennifer Rexford & David Walker. *An Assertion Language for Debugging SDN Applications* in *Proceedings of the 3rd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking HotSDN '14* (ACM, Chicago, Il, USA, 2014) (cited on page 83).

120. Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis & Scott Shenker. *Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '14* (ACM, Chicago, IL, USA, 2014) (cited on pages 83, 108, 109, 111).

121. Arjun Guha, Mark Reitblatt & Nate Foster. *Machine-verified Network Controllers* in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '13* (ACM, Seattle, WA, USA, 2013) (cited on page 84).

122. J. Christian" Attiogbé. *Building Correct SDN Components from a Global Event-B Formal Model* in *Formal Aspects of Component Software* (Springer International Publishing, Cham, 2018) (cited on page 84).

123. Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić & Jennifer Rexford. *A NICE Way to Test OpenFlow Applications* in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI '12, San Jose, CA, USA* (USENIX Association, 2012) (cited on page 84).

124. Rupak Majumdar, Sai Deep Tetali & Zilong Wang. *Kuai: A Model Checker for Software-defined Networks* in *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design FMCAD '14* (IEEE, Lausanne, Switzerland, 2014) (cited on page 84).

125. Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira & Asaf Valadarsky. *VeriCon: Towards Verifying Controller Programs in Software-defined Networks* in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '14* (ACM, Edinburgh, UK, 2014) (cited on pages 84, 122).

126. Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang & Guofei Gu. *Attacking the Brain: Races in the SDN Control Plane* in *Proceedings of the 26th USENIX Security Symposium USENIX Security '17* (USENIX Association, Vancouver, BC, Canada, 2017) (cited on page 84).

127. Jiangyuan Yao, Zhiliang Wang, Xia Yin, Xingang Shi, Yahui Li & Chongrong Li. *Testing Black-Box SDN Applications with Formal Behavior Models* in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2017), 110 (cited on page 84).

128. Elvira Albert, Miguel Gómez-Zamalloa, Albert Rubio, Matteo Sammartino & Alexandra Silva. *SDN-Actors: Modeling and Verification of SDN Programs* in *International Symposium on Formal Methods* (2018), 550 (cited on page 84).

129. Lalita J Jagadeesan & Veena Mendiratta. *Analytics-Enhanced Automated Code Verification for Dependability of Software-Defined Networks* in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2017), 139 (cited on page 84).

130. Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger & David Walker. *Abstractions for Network Update* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '12* (ACM, Helsinki, Finland, 2012) (cited on pages 85, 89, 102).

131. Jedidiah McClurg, Hossein Hojjat, Pavol Černý & Nate Foster. *Efficient Synthesis of Network Updates* in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '15* (ACM, Portland, OR, USA, 2015) (cited on page 85).

132. Naga Praveen Katta, Jennifer Rexford & David Walker. *Incremental Consistent Updates* in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking HotSDN '13* (ACM, Hong Kong, China, 2013) (cited on page 85).

133. Jedidiah McClurg, Hossein Hojjat & Pavol Černý. *Synchronization Synthesis for Network Programs* in *Proceedings of the 29th International Conference on Computer Aided Verification CAV '17* (Springer, Heidelberg, Germany, 2017) (cited on page 85).

134. Mark Reitblatt, Marco Canini, Arjun Guha & Nate Foster. *FatTire: Declarative Fault Tolerance for Software-Defined Networks* in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking HotSDN '13* (ACM, Hong Kong, China, 2013) (cited on page 85).

135. Xiaoye Sun, Apoorv Agarwal & Tze Sing Eugene Ng. *Attendre: Mitigating Ill Effects of Race Conditions in Openflow via Queueing Mechanism* in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (ACM, New York, NY, USA, 2012), 137 (cited on page 85).

136. Andreas Wundsam, Dan Levin, Srini Seetharaman & Anja Feldmann. *OFRewind: Enabling Record and Replay Troubleshooting for Networks* in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference USENIXATC '11* (USENIX Association, Portland, OR, USA, 2011) (cited on page 85).

137. Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres & Nick McKeown. *Where is the Debugger for My Software-defined Network?* in *Proceedings of the 1st ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking HotSDN '12* (ACM, Hong Kong, China, 2012) (cited on page 85).

138. Pavol Bielik, Veselin Raychev & Martin Vechev. *Scalable Race Detection for Android Applications* in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA'15* (ACM, Pittsburgh, PA, USA, 2015) (cited on page 85).

139. Woosuk Lee, Wonchan Lee & Kwangkeun Yi. *Sound Non-Statistical Clustering of Static Analysis Alarms* in *International Workshop on Verification, Model Checking, and Abstract Interpretation* (2012) (cited on page 85).

140. T. Muske, A. Baid & T. Sanas. *Review Efforts Reduction by Partitioning of Static Analysis Warnings* in *Source Code Analysis and Manipulation (SCAM)* (IEEE, 2013) (cited on page 85).

141. T. Muske. *Improving Review of Clustered-Code Analysis Warnings* in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution ICSME* (IEEE, Victoria, BC, Canada, 2014) (cited on page 85).

142. Wei Le & Mary Lou Soffa. *Path-based Fault Correlations* in *Proceedings of the ACM SIGSOFT 18th International Symposium on the Foundations of Software Engineering FSE '10* (ACM, Santa Fe, NM, USA, 2010) (cited on page 85).

143. Tukaram Muske & Alexander Serebrenik. *Survey of Approaches for Handling Static Analysis Alarms* in *Proceedings of 16th International Working Conference on Source Code Analysis and Manipulation* (IEEE, 2016) (cited on page 85).

144. Baris Kasikci, Cristian Zamfir & George Candea. *Data Races vs. Data Race Bugs: Telling the Difference with Portend* in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS XVII* (ACM, London, UK, 2012) (cited on page 85).

145. *Open vSwitch. Production Quality, Multilayer Open Virtual Switch* `http://openvswitch.org/` (cited on pages 87, 95, 98).

146. Maciej Kuźniar, Peter Perešíni & Dejan Kostić. *What You Need to Know About SDN Flow Tables* in *Proceedings of the 16th International Conference on Passive and Active Measurement PAM '15* (Springer, New York, NY, USA, 2015) (cited on pages 89, 97).

147. Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood & Andrew W. Moore. *OFLOPS: An Open Framework for Openflow Switch Evaluation* in *Proceedings of the 13th International Conference on Passive and Active Measurement PAM '12* (Springer, Vienna, Austria, 2012) (cited on pages 89, 97).

148. Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, *et al.* *B4: Experience with a Globally-Deployed Software Defined WAN* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '13* (ACM, Hong Kong, China, 2013) (cited on pages 89, 97).

149. *Floodlight Open SDN Controller* `http://projectfloodlight.org/floodlight` (cited on pages 90, 110).

150. James Mccauley. *POX: A Python-based OpenFlow Controller* `https://github.com/noxrepo/pox` (cited on pages 90, 110).

151. Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris & Eddie Kohler. *The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors* in *Proceedings of the 24th Symposium on Operating Systems Principles SOSP '13* (ACM, Farmington, PA, USA, 2013) (cited on page 98).

152. Dimitar Dimitrov, Veselin Raychev, Martin Vechev & Eric Koskinen. *Commutativity Race Detection* in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '14* (ACM, Edinburgh, UK, 2014) (cited on pages 98, 102, 103).

153. Anduo Wang, Wenchao Zhou, Brighten Godfrey & Matthew Caesar. *Software-Defined Networks as Databases* in *the Open Networking Summit 2014 (ONS 2014)* (USENIX Association, Santa Clara, CA, 2014) (cited on page 102).

154. Ratul Mahajan & Roger Wattenhofer. *On Consistent Updates in Software Defined Networks* in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013) (cited on page 102).

155. László Babai. Graph Isomorphism in Quasipolynomial Time. *Computing Research Repository (CoRR)* **abs/1512.03547** (2015) (cited on page 105).

156. Derek G. Corneil & David G. Kirkpatrick. A Theoretical Analysis of Various Heuristics for the Graph Isomorphism Problem. *SIAM Journal on Computing* (1980) (cited on page 105).

157. Anil K. Jain & Richard C. Dubes. *Algorithms for Clustering Data* (Prentice-Hall, Inc., 1988) (cited on page 108).

158. Veselin Raychev, Martin Vechev & Manu Sridharan. *Effective Race Detection for Event-driven Programs* in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA'13* (ACM, Indianapolis, IN, USA, 2013) (cited on pages 113, 114).

159. Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah & Hans Fugal. *Fastpass: A Centralized "Zero-Queue" Datacenter Network* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '14* (ACM, Chicago, IL, USA, 2014) (cited on page 117).

160. Santhosh Prabhu, Gohar Irfan Chaudhry, Brighten Godfrey & Matthew Caesar. *High-Coverage Testing of Softwarized Networks* in *Proceedings of the ACM SIGCOMM 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges SecSoN '18* (ACM, Budapest, Hungary, 2018) (cited on page 122).

161. Glenn Ammons, Rastislav Bodik & James R. Larus. *Mining Specifications* in *Proceedings of the 29th ACM SIGPLAN Symposium on Principles of Programming Languages POPL '02* (ACM, Portland, OR, Oregon, 2010) (cited on page 122).

162. Mark Gabel & Zhendong Su. *Testing Mined Specifications* in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering FSE '12* (ACM, Cary, North Carolina, 2012) (cited on page 122).

163. David Lo, Siau-Cheng Khoo & Chao Liu. *Mining Past-time Temporal Rules from Execution Traces* in *Proceedings of the 2008 International Workshop on Dynamic Analysis WODA '08* (ACM, Seattle, Washington, 2008) (cited on page 122).

CURRICULUM VITAE

PERSONAL DATA

Name	Ahmed Elhassany
Date of Birth	April 1, 1986
Citizen of	Palestine

EDUCATION

2015 – 2019	*Ph.D. Student*, ETH Zürich Zürich Switzerland
2009 – 2011	*M.S. Computer Science*, University of Delaware Newark, DE, USA.
2003 – 2008	*B.Sc. Computer Engineering*, Islamic University of Gaza Gaza, Palestine.

EMPLOYMENT

June '15 – Oct. '19	Research Assistant *ETH Zürich* Zürich, Switzerland
June – Sept. '18	Software Engineer Intern *Facebook Inc.* Menlo Park, CA, USA
Spring '15	Research Associate *Indiana University* Bloomington, IN, USA
July '13 – Nov. '14	Research Scientist *International Computer Science Institute (ICSI)* Berkeley, CA, USA

May – July ’13	Summer Student *Lawrence Berkeley National Lab. (LBL)* Berkeley, CA, USA
Aug. ’11 – July ’13	Research Associate *Indiana University* Bloomington, IN, USA
August ’10	Summer Student *Lawrence Berkeley National Lab. (LBL)* Berkeley, CA, USA
Feb. ’09 – July ’09	Software Engineer *Municipality of Gaza* Gaza, Palestine
Dec. ’08 – Mar. ’09	Independent Consultant *Palestinian National Internet Naming Authority (PNINA)* Gaza, Palestine
June – Sept ’08	Student Developer *Google Inc. & Internet2* Gaza, Palestine
Sep. ’07 – Mar. ’08	Software Engineer *AfkarIT* Gaza, Palestine
Sep. ’04 – Oct. ’05	Contractor Software Engineer *Ard El-Insan* Gaza, Palestine