Life on the Edge: Unraveling Policies into Configurations

Shrutarshi Basu
Cornell University

Nate Foster
Cornell University

Hossein Hojjat
RIT

Paparao Palacharla
Fujitsu Labs of America

Christian Skalka
University of Vermont

Xi Wang
Fujitsu Labs of America

Abstract

Current frameworks for network programming assume that the network comprises a set of homogenous devices that can be rapidly reconfigured in response to changing policies and network conditions. Unfortunately, these assumptions are incompatible with the realities of modern networks, which contain legacy devices that offer diverse functionality and can only be reconfigured slowly. Additionally, network service providers need to walk a fine line between providing flexibility to users, and maintaining the integrity and reliability of their core networks. These issues are particularly evident in optical networks, which are used by ISPs and WANs and provide high bandwidth at the cost of limited flexibility and long reconfiguration times. This paper presents a different approach to implementing high-level policies, by pushing functionality to the edge and using the core merely for transit. Building on the NetKAT framework and leveraging linear programming solvers, we develop techniques for analyzing and transforming policies into configurations that can be installed at the edge of the network. Furthermore, our approach can be extended to incorporate constraints that are crucial in the optical domain, such as path constraints. We develop a working implementation using off-the-shelf solvers and evaluate our approach on realistic optical topologies.

1. INTRODUCTION

Recent years have seen the development and deployment of increasingly programmable network devices, as well as software systems for managing them. Advances in this area fall under the general umbrella of *Software-Defined Networking* (SDN) and include new frameworks for managing network configurations [6, 21, 20, 16], new languages for programming networks [10, 34, 33, 5, 27], and new applications that implement advanced network functionality [15, 14]. However, most existing SDN frameworks assume that the network comprises a collection of homogenous devices that can be rapidly reconfigured as policies change or network conditions evolve—an assumption that is often unrealistic in practice:

- **Legacy Devices:** Many organizations deploy thousands of network devices that are supplied by multiple different vendors. SDN deployments, where they exist, tend to be partial in nature. Hence, any framework for real-world network programming must be able to implement high-level policies in the presence of legacy devices.
- **Heterogeneous Functionality:** There is a fundamental mismatch between the capabilities of SDN switches, which allow packets to be transformed in essentially arbitrary ways at each hop, and devices such as IP routers, MPLS LSRs, and optical ROADMs. It is not clear how to incorporate these devices and their specific limitations into SDN-like programming frameworks, which assume a homogeneous collection of switches.
- **Performance Limitations:** Existing frameworks tacitly assume that it is possible to rapidly reconfigure devices in response to policy updates, traffic shifts, topology changes, and other changes. However, on many devices, updating a forwarding table can take several seconds, which limits the network's ability to react to changing conditions.

Taken together, these issues strongly suggest that a fundamentally different model for network programming is needed.

One way to address these challenges is to distinguish the "edge" devices at the perimeter of the network from the "fabric" devices in the core that connect edge devices to each other. So long as the edge devices provide SDN-like functionality, it is possible to implement a broad set of policies specified by programmers. Meanwhile the fabric devices, which are more constrained, only need to implement the "plumbing" needed to carry packets across the network. By carefully enforcing a division of labor between edge and fabric devices, it is possible to address all three of the issues discussed above. Heterogeneous devices in the core of the network are only required to implement infrequently changing configurations as part of the fabric, while updates to policy can be realized at the edge, reducing overhead.

The distinction between edge and fabric devices is particularly relevant in optical circuit networks that are used to connect traditional packet networks. In these networks, traffic is transported using optical channels which are slow to set up (on the order of several seconds), but allow for high bandwidth, at the cost of flexibility. Unlike packet networks, where the header fields of each packet can be used to control forwarding, optical forwarding devices typically can only forward according to the frequency range occupied by a channel, and cannot easily access the header fields of the transported packets. Extending the flexibility of SDN to these networks would be a great improvement to their current level of programmability. For these reasons, we focus on optical circuit networks as our primary application domain.

This paper presents the design and implementation of a practical framework for implementing high-level policies at the edge, using a

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

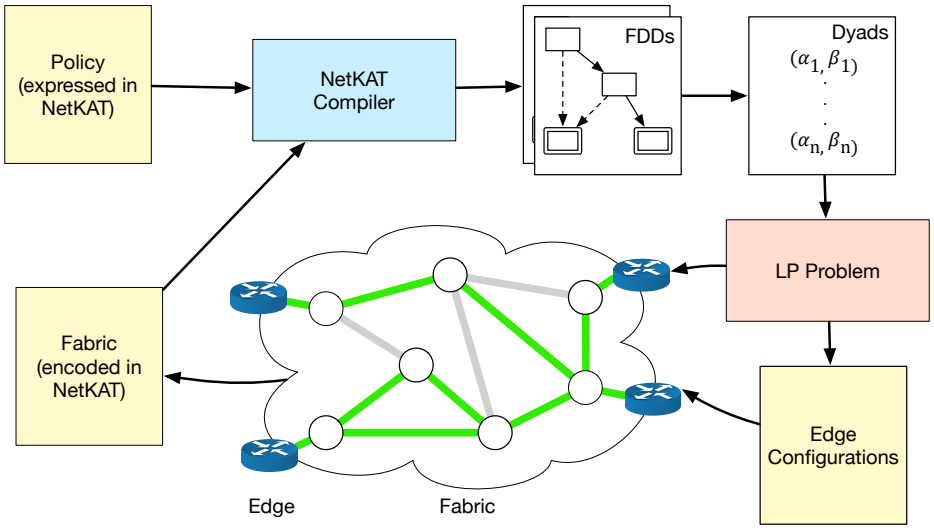


Figure 1: System architecture.

fixed fabric to provide connectivity through the core of the network. The idea of implementing network programs in terms of an edge-fabric distinction is not new. In fact, the term "fabric" is borrowed from a paper by Casado et al. that is motivated by some of the same issues identified above [7]. Recent work on the Felix traffic measurement system also adopts an edge-fabric distinction: it analyzes the paths used by the fabric and pushes monitoring tasks to the edge of the network [8]. Panopticon tackles the issue of partial SDN deployments in enterprise networks, but does not distinguish between edge and fabric devices [23].

We had to overcome a number of technical challenges in building our framework for "unraveling" policies into configurations:

- **Analysis:** To generate configurations for the devices at the edge, we need to determine how the fabric forwards traffic between end points. We exploit recent advances in data plane verification, and build on the NetKAT framework [2], to compute the requirements of the policy and the forwarding paths provided by the fabric.
- **Adaptation:** To faithfully implement the functionality specified by a high-level policy using a fixed fabric, we need to check that the transformations on packets performed in the fabric are not in conflict with the transformations performed at the edge. Sometimes it is possible to co-opt "spare" bits in the header field to encode the high-level policy, but more generally it is necessary to rely on some form of tunneling.
- **Expressiveness:** Certain policies can be expressed in terms of a "one big switch" abstraction, in which only input-output behavior matters [18]. But other policies, such as network function virtualization and middlebox service chaining require paths with multiple segments, or require that paths traverse certain nodes. Our approach naturally supports policies based on the "one big switch" abstraction, and we add natural extensions to support segmented paths and path constraints.

We have built a prototype implementation of our approach for rewriting high-level policies into edge device configurations. The front-end analyzes the high-level policy and fabric configuration using NetKAT. The artifacts of this analysis are then supplied to a back-end that encodes them into constraint problems that can be solved using a linear programming (LP) solver. The solution provided by the solver is finally translated into forwarding rules that are installed on the SDN-enabled edge switches at the edge. To evaluate the performance and scalability of our approach, we conducted experiments using real-world topologies and synthetic configurations.

The contributions of this paper are as follows:

1. We develop a practical framework for implementing high-level network policies, based on the "one big switch" abstraction. Our framework supports heterogeneous devices, focusing on networks with optical circuit cores and SDN-enabled packet switches at edge.
2. We show how to automatically analyze and transform policies into equivalent edge configurations using the NetKAT framework and off-the-shelf solvers for LP problems.
3. We design extensions to our basic framework including: (i) multi-segment paths supporting applications like service chaining and network function virtualizaion, and (ii) path constraints, which gives policies finer-grained control over how the fabric is utilized.
4. We present an implementation that supports heterogenous networks, using SDN devices for the edge and optical circuit devices for the core. We evaluate our implementation on a large-scale, real-world topology and synthetic policies.

The rest of this paper is organized as follows: Section 2 motivates the edge-fabric distinction and informally explains our approach using an example of an optical circuit network. Section 3 explores the space of fabrics and policies and positions our work. Section 4 describes the NetKAT programming language, focusing on the properties we leverage, and extensions to support optical networks. Section 5 describes the analysis techniques we build atop the NetKAT compiler, and the translation to linear programming problems. Sections 6 details our implementation and 7 evaluates it on a real-world optical topology. Section 8 discusses related work and Section 9 explores possible future directions. We conclude in Section 10.

2. BACKGROUND

This section presents our approach, using a simple running example. Figure 1 depicts the individual components of our system and its overall architecture.

2.1 Optical Networks

The distinction between edge and fabric is particularly prominent in packet-optical networks, where the edge is composed of flexible electrical switches, and the fabric is a optical circuit network.

In the optical core, network nodes are connected to each other via optical fibers. Information is transmitted using *wave-division multiplexing* (WDM), which allows multiple *channels* to occupy a single fiber on different frequency slots. WDM requires a multiplexer at the source to combine multiple optical channels and a demutiplexer at the destination to separate them back out. These functions can be combined into a single physical device called an *optical add-drop multiplexer* (OADM). OADMs typically have multiple ports, each connecting an optical fiber to the OADM. On *reconfigurable optical add-drop multiplexers* (ROADMs), the optical channels that are demultiplexed from a particular port can be directed to different outgoing ports, via software reconfiguration. In the rest of this paper, we assume all devices in our optical core are ROADMs.

The edge of the network is composed of electrical switches, which are connected to ROADMs via transceiver ports. When an edge switch sends data to the optical core, the transceiver converts the incoming electrical signals to an optical signal occupying a single optical channel. The multiplexer on the ingress ROADM combines several such single channels and emits the resulting signal on a physical optical fiber. Conversely, on the receiver side, the egress ROADM's multiplexer separates out the optical signal from a physical fiber into the constituent channels, and emits each channel to a transceiver on a particular port. The transceiver converts the optical signal to an electrical signal for transmission to the receiving host.

We call the optical channel connecting an ingress and egress ROADM a *lightpath*—a single-source, single-sink channel that may span multiple contiguous fiber links, but must occupy the same set of frequency slots on each such link. Each ROADM in a multi-link lightpath demultiplexes incoming optical signals, routes the separated channels to possibly different ports and then multiplexes the signals for each port on to the connected optical fiber.

Crucially, reconfiguring an optical lightpath may take several seconds, as opposed to packet switches which can be reconfigured in milliseconds. This has important repercussions for network management, especially for latency critical applications and fault tolerance. To quickly respond to policy changes, a network management solution must minimize reconfigurations of optical lightpaths. Thus, it is imperative that most changes be implemented using the packet switches at the edge. The framework developed in this paper addresses this need, allowing the network administrator to flexibly change policies while restricting configuration changes to the edge.

2.2 Example

As an example of a hybrid packet-optical network, consider the topology shown in Figure 2. It consists of a fabric of ROADMs (diamonds), SDN-enabled packet switches on the edge (circles) and end-hosts connected to the switches (squares). Let us assume that there are two optical channels set up, one connecting port 2 on ROADM 4 to port 2 on ROADM 5, and another connecting port 3 on ROADM 4 to port 2 on ROADM 6.

A network user may write a variety of policies that make use of this topology. One possible policy is that all SSH traffic (TCP destination port 22) from host 1 is sent only to host 3, and all Web traffic from host 1 is sent only to host 2. To express this policy, we need a concise way of specifying predicates on packet header fields, and forwarding paths through the network. Fortunately, the NetKAT programming language allows us to do just this [2]. A NetKAT program allows a user to express network behavior in terms of functions on packets. These programs are implemented in the network

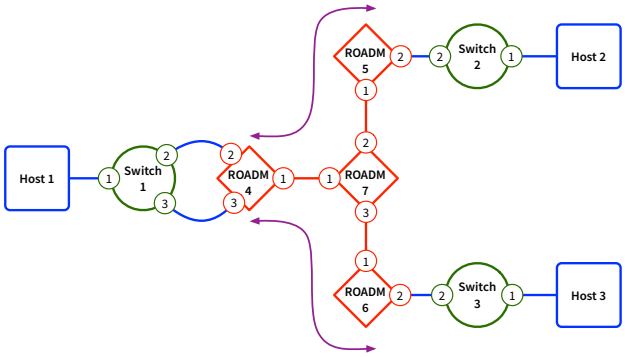


Figure 2: Optical fork topology.

Listing 1: User Policy NetKAT Program.

```
if switch=1 and port=1 and tcpDst=80 then
  (s1:1 => s2:1)
else if switch=1 and port=1 and tcpDst=22 then
  (s1:1 => s3:1)
else if switch=2 and port=1 then
  (s2:1 => s1:1)
else if switch=3 and port=1 then
  (s3:1 => s1:1)
```

Listing 2: Optical Configuration NetKAT Program.

```
if roadm=4 and port=2 then
  channel:=1; (r4:1 => r7:1); (r7:2 => r5:1);
  port:=2
else if roadm=4 and port=3 then
  channel:=2; (r4:1 => r7:1); (r7:3 => r6:1);
  port:=2
else if roadm=5 and port=2 then
  channel:=1; (r5:1 => r7:2); (r7:1 => r4:1);
  port:=2
else if roadm=6 and port=2 then
  channel:=2; (r6:1 => r7:3); (r7:1 => r4:1);
  port:=3
```

by compiling them to flowtables for SDN-capable switches (eg., switches supporting the OpenFlow protocol). Section 4 describes the syntax and semantics of the language, as well as the process for compiling policies to flowtables. The intended forwarding behavior in this example is concisely expressed by the NetKAT program shown in Listing 1.

Implementing this policy on the given topology presents several challenges. Since reconfiguring the optical channels incurs a large time penalty, we should avoid changing the existing core configuration if possible. In that case, even though the user is only concerned with end-to-end behavior, they would have to understand the details of the optical fabric configuration. Then, instead of the clean policy shown in Figure 1, the user would have to manually match up packet switch ports to the optical transceiver ports and the channels they connect to. Finally, the user would need to write another program—either in NetKAT or directly as a forwarding table—that operates on the edge switches and correctly implements the desired forwarding behavior. Although doing all this is feasible in principle, it is quite tedious and error prone process—moreover, it would need to be repeated every time the policy changes.

Fortunately, in addition to allowing us to specify network behavior, the NetKAT language and its compiler provides the tools required to automate this rewriting. First, we encode the configu-

Listing 3: Generated NetKAT Ingress Program.

```
if switch=1 and port=1 and tcpDst=80 then
  vlanId := 1; port := 2
else if switch=1 and port=1 and tcpDst=22 then
  vlanId := 2; port := 3
else if switch=2 and port=1 then
  vlanId := 3; port := 2
else if switch=3 and port=1 then
  vlanId := 4; port := 2
```

Listing 4: Generated NetKAT Egress Program.

```
if vlanId=1 and switch=2 and port=2 then
  strip vlan; port := 1
else if vlanId=2 and switch=3 and port=2 then
  strip vlan; port := 1
else if vlanId=3 and switch=1 and port=2 then
  strip vlan; port := 1
else if vlanId=4 and switch=1 and port=3 then
  strip vlan; port := 1
```

ration of the optical core in NetKAT as well, using a subset of the language that can be compiled to optical circuits (described in Section 4.2). Listing 2 shows a NetKAT program that implements the optical channels for this example. Using this fabric program, we can "unravel" the user policy into ingress and egress NetKAT programs that apply only to the edge switches (as shown in Listings 3 and 4 respectively). Section 5 describes how we use the NetKAT compiler to transform programs into pairs of predicates and modifications that capture the input-output behavior of the network, and then match the pairs from the policy with those in the fabric.

The user can treat the core fabric as "one big switch" that connects the edge locations. Our compiler finds paths in the fabric that provide the end-to-end connectivity required by the policy, and relocates any predicates (eg., `tcpDst=80`) or modifications specified in the policy to the edge. Changes to the policy only affect the edge switches. This reduces the overheads associated with implementing policy changes, assuming the fabric changes infrequently.

To perform this rewriting, we leverage the intermediate data structures used in the NetKAT compiler. NetKAT programs are compiled to intermediate representations called Forwarding Decision Diagrams (FDDs), which are used to generate forwarding tables for switches running the OpenFlow protocol [29]. By analyzing the FDD representation of the program we can pair up each predicate used to distinguish traffic classes, and the corresponding modifications made to matching packets. We call each such pair a *dyad*. Since NetKAT treats switch and port location as logical fields in packet headers, the collected predicates include the starting switch and port, and modifications include the destination switch and port. Thus each pair of a predicate and modification denotes the source and sink of a particular traffic class.

Performing this analysis on the user policy gives us the *required* sources and sinks of each traffic class. The same analysis on the fabric gives us sources and sinks of each path *provided* by fabric. We develop two back-ends to match the required end-points to the provided paths—one using simple graph algorithms, and another using a translation to a linear programming problem, which can be solved automatically.

Once policy end-points are matched to fabric paths, we need to distinguish multiple policy traffic classes that are sent across the same fabric-provided paths. At the destination, we may need to separate them out again, either to forward out different ports, or to modify header fields in different ways. This separation is achieved by generating an ingress program that uses the policy's predicates to match incoming traffic, apply a unique tag to each packet and forward to ports that match suitable paths in the fabric (as in Listing 3). Conversely, the egress program matches on the tag at edge locations and performs any policy-specified modifications and final forwarding (as in Listing 4).

3. FABRIC REQUIREMENTS

The example presented in Section 2 uses an optical fabric that provides connectivity between two end-points, with no further restrictions or modification on what kind of traffic it carries. However, in general, the fabric may only forward certain traffic classes—i.e., it may require incoming traffic to satisfy some other predicate in order to be forwarded to its destination. The fabric may also modify packets in transit in ways that conflict with the policy. For example, if the fabric forwards packets based on destination addresses, then incoming packets will need to have a particular address to reach the correct destination. Hence, traffic entering at the edge of the network may need to be modified to reach the correct destination, and these modifications should be reverted at the end. In general, any approach to rewriting a network policy to utilize an existing fabric will depend on properties of the fabric and policies, in terms of forwarding ability, predicate enforcement and underlying implementation. This suggests a rich space of fabric capabilities and policy requirements that is worth classifying to guide further development.

In order to implement a given policy, the fabric must satisfy two crucial requirements. First, it must provide connectivity between the required end-points. Second, it must not irreversably alter traffic that passes through it. Section 5.2 describes a *path matching* technique that tests for the first condition—whether a fabric provides the required connectivity. Since multiple classes of policy traffic may traverse the same fabric path, utilizing these connections requires some form of tagging in general (possibly a VLAN tag or MPLS label) to keep the policy-defined traffic classes separate. The modifications required by the policy can be extracted from the FDD representation of a NetKAT program, and can be applied at the destination by checking the tag. These techniques can be implemented using switches supporting SDN protocols like OpenFlow.

If the fabric modifies packet headers, then even if policy traffic can be directed to the correct destination, it might be modified in a way that violates the policy (for example, a fabric that performs some kind of network address translation would break any policy that depends on IP addresses and TCP ports). Having some form of tunneling available at the end points avoids this problem.

An adequate tunneling mechanism encapsulates incoming packets, protecting them from modifications made by the fabric, and allows recovery of the original, unaltered packet at the destination. As far as the policy is concerned, traffic is carried unchanged from one point on the edge to another. This could be provided by protocols such as MPLS or VXLAN, or by custom dataplanes written using a language like P4 [5]. Tunneling is common enough that we can safely assume some form of tunneling to be present in practical network deployments.

The techniques described in this paper focus on fabrics implemented using optical channels. In this setting, connectivity between two points is provided by an optical channel between them. Optical fabrics provide a form of tunneling: ROADMs can operate only on the optical signals without affecting the packets they carry. They provide the necessary connectivity and tunneling properties, and provide good motivation for exploring the edge-fabric distinction (as described in Section 2).

If a fabric provides connectivity, but does not have some form of encapsulation, all is not lost—the fabric may test header fields on

Syntax

Naturals	n	$::=$	$0 \mid 1 \mid 2 \mid \ldots$
Fields	f	$::=$	$f_1 \mid \cdots \mid f_k$
Packets	pk	$::=$	$\{f_1 = n_1, \cdots, f_k = n_k\}$
Predicates	a, b	$::=$	$true$ *Identity*
		$\mid$	$false$ *Drop*
		$\mid$	$f{=}n$ *Test*
		$\mid$	$a + b$ *Disjunction*
		$\mid$	$a \cdot b$ *Conjunction*
		$\mid$	$\neg a$ *Negation*
Programs	p, q	$::=$	a *Filter*
		$\mid$	$f{\leftarrow}n$ *Modification*
		$\mid$	$p + q$ *Union*
		$\mid$	$p \cdot q$ *Sequencing*
		$\mid$	p^* *Iteration*

Semantics

$$[\![p]\!] \in pk \to \{pk\}$$
$$[\![true]\!]\ pk \triangleq \{pk\}$$
$$[\![false]\!]\ pk \triangleq \{\}$$
$$[\![f{=}n]\!]pk \triangleq \begin{cases} \{pk\} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases}$$
$$[\![\neg a]\!]\ pk \triangleq \{pk\} \setminus ([\![a]\!]\ pk)$$
$$[\![f{\leftarrow}n]\!]\ pk \triangleq \{pk[f := n]\}$$
$$[\![p + q]\!]\ pk \triangleq [\![p]\!]\ pk \cup [\![q]\!]\ pk$$
$$[\![p \cdot q]\!]\ pk \triangleq ([\![p]\!] \bullet [\![q]\!])\ pk$$
$$[\![p^*]\!]\ pk \triangleq \bigcup_i F^i\ pk$$
where $F^0\ pk \triangleq \{pk\}$ and $F^{i+1}\ pk \triangleq ([\![p]\!] \bullet F^i)\ pk$

Figure 3: NetKAT abstract syntax and semantics.

Filter	f	$::=$	$\text{switch} = n \cdot \text{port} = n$
Channel	w	$::=$	$\text{channel} := n$
Links	l	$::=$	$sw : pt \to sw' : pt'$
		$\mid$	$l \cdot l$
Circuit	c	$::=$	$f \cdot\ w \cdot\ l \cdot\ \text{port} := n$
Programs	p	$::=$	c
		$\mid$	$c + c$

(a) Circuit NetKAT syntax.

Allocation	$A \in (sw, pt)$	$\to$	channel
Channel	$C \in \text{circuit}$	$\to$	channel
Path	$P \in \text{circuit}$	$\to$	$\{(sw, pt)\}$

$$\frac{\begin{array}{c} f = \text{switch} = sw \cdot \text{port} = pt \\ w = \text{channel} := n \\ c = f \cdot\ w \cdot\ l \cdot\ \text{port} := pt' \\ \text{last}(c) = (sw', pt') \\ \forall (sw, pt) \in P(c).\ A(sw, pt) = n \end{array}}{A \vdash c}\ \textsc{Continuity}$$

$$\frac{\begin{array}{c} A \vdash c_1 \qquad A \vdash c_2 \\ \text{C}(c_1) = \text{C}(c_2) \Rightarrow \text{P}(c_1) \cap \text{P}(c_2) = \varnothing \end{array}}{A \vdash c_1 + c_2}\ \textsc{Disjointness}$$

(b) Circuit NetKAT validity rules.

Figure 4: Circuit NetKAT syntax and validity rules.

incoming traffic, or make modifications before it exits the fabric. If the fabric tests header fields, but performs no modifications, we can still use path matching to detect appropriate fabric paths. The same technique that we use to detect a fabric path between two end points also detects the complete predicate on packet headers tested by the fabric to take that path. Even if this tests are applied in the middle of the fabric, or is distributed across multiple nodes, the technique described in Section 5.2 can collect and "pull out" this predicate so that it can be used for analysis as described below. Alternatively, the fabric may conflict with the policy. This conflict can be detected by analyzing the fabric's requirements according to three cases:

1. The fabric tests only switch and port. In this case, no modifications are required, traffic only has to be directed to the correct location to enter the fabric.

2. The fabric tests a subset of fields that a policy predicate tests. In this case, the policy already requires us to produce a fine-grained forwarding rule to enter the fabric. The ingress rule can apply the packet header modifications to satisfy the fabric as well as a unique tag. The egress program matches this tag and rewrites the modified headers back to what the policy expects. Since we generate rules to attach and match tags anyway, this does not increase the size of flowtables on the edge switches. This is also handled easily by SDN-capable switches.

3. For all other cases, it is necessary to encapsulate the user traffic in a packet whose headers may be modified freely to satisfy the fabric and then discarded at the egress.

However, if the fabric alters incoming traffic in some way, there is no general way to reverse the modifications (at least, without an exponential increase in the size of the flow tables at the edge). Thus a lack of encapsulation is not fatal, but severely restricts what classes of fabrics can be targeted by our techniques.

4. NETKAT LANGUAGE

This section reviews the syntax and semantics of NetKAT, to provide background for the rest of the paper. NetKAT is a domain-specific language for specifying and reasoning about packet forwarding behavior in SDN-enabled networks [2, 11, 29]. The language provides high-level, hardware-independent constructs for writing network programs, as well as sound and complete mechanisms for reasoning about the packet-forwarding behavior.

The abstract syntax and semantics of NetKAT are presented in Figure 3. The semantics of the NetKAT language is given in terms of functions on packets. The notation $[\![p]\!]\ pk = S$ means that program p produces output set S when supplied with input packet pk. For readers familiar with NetKAT, note that we only model the *local* semantics of the language—i.e., we do not keep track of packet histories, as we do not need them in this paper.

NetKAT is higher-level than the APIs exposed by most SDN controllers, which are usually based around low-level structures like forwarding tables for switches. By contrast, NetKAT programs are functions defined in terms of predicates on packet header fields, modifications to the headers, and combinations thereof using sequencing, union and iteration operators.

$\boxed{d_1 + d_2}$

$$\{a_{11},\ldots,a_{1k}\} + \{a_{21},\ldots,a_{2l}\} \triangleq \{a_{11},\ldots,a_{1k}\} \cup \{a_{21},\ldots,a_{2l}\}$$

$$(f{=}n\,?\,d_{11} : d_{12}) + \{a_{21},\ldots a_{2l}\} \triangleq (f{=}n\,?\,d_{11} + \{a_{21},\ldots a_{2l}\} : d_{12} + \{a_{21},\ldots a_{2l}\})$$

$$(f_1{=}n_1\,?\,d_{11} : d_{12}) + (f_2{=}n_2\,?\,d_{21} : d_{22}) \triangleq \begin{cases} (f_1{=}n_1\,?\,d_{11} + d_{21} : d_{12} + d_{22}) & \text{if } f_1 = f_2 \text{ and } n_1 = n_2 \\ (f_1{=}n_1\,?\,d_{11} + d_{22} : d_{12} + (f_2{=}n_2\,?\,d_{21} : d_{22})) & \text{if } f_1 = f_2 \text{ and } n_1 \sqsubset n_2 \\ (f_1{=}n_1\,?\,d_{11} + (f_2{=}n_2\,?\,d_{21} : d_{22}) : d_{12} + (f_2{=}n_2\,?\,d_{21} : d_{22})) & \text{if } f_1 \sqsubset f_2 \end{cases}$$

(omitting symmetric cases)

$\boxed{d|_{f=n}}$

$$\{a_1,\ldots,a_k\}|_{f=n} \triangleq (f{=}n\,?\,\{a_1,\ldots,a_k\} : \{\})$$

$$(f_1{=}n_1\,?\,d_{11} : d_{12})|_{f=n} \triangleq \begin{cases} (f{=}n\,?\,d_{11} : \{\}) & \text{if } f = f_1 \text{ and } n = n_1 \\ (d_{12})|_{f=n} & \text{if } f = f_1 \text{ and } n \neq n_1 \\ (f{=}n\,?\,(f_1{=}n_1\,?\,d_{11} : d_{12}) : \{\}) & \text{if } f \sqsubset f_1 \\ (f_1{=}n_1\,?\,(d_{11})|_{f=n} : (d_{12})|_{f=n}) & \text{otherwise} \end{cases}$$

$\boxed{d_1 \cdot d_2}$

$$a \cdot \{a_1,\ldots,a_k\} \triangleq \{a \cdot a_1,\ldots,a \cdot a_k\}$$

$$a \cdot (f{=}n\,?\,d_1 : d_2) \triangleq \begin{cases} a \cdot d_1 & \text{if } f{\leftarrow}n \in a \\ a \cdot d_2 & \text{if } f{\leftarrow}n' \in a \wedge n' \neq n \\ (f{=}n\,?\,a \cdot d_1 : a \cdot d_2) & \text{otherwise} \end{cases}$$

$$\{a_1,\ldots,a_k\} \cdot d \triangleq a_1 \cdot d + \ldots + a_k \cdot d$$

$$(f{=}n\,?\,d_{11} : d_{12}) \cdot d_2 \triangleq (d_{11} \cdot d_2)|_{f=n} + (d_{12} \cdot d_2)|_{f\neq n}$$

$\boxed{\neg d}$

$$\neg\{\} \triangleq \{\{\}\}$$

$$\neg\{a_1,\ldots,a_k\} \triangleq \{\} \text{ where } k \geq 1$$

$$\neg(f{=}n\,?\,d_1 : d_2) \triangleq (f{=}n\,?\,\neg d_1 : \neg d_2)$$

$\boxed{d*}$

$$d* \triangleq \mathsf{fix}\,(\lambda d'.\ \{\{\}\} + d \cdot d')$$

Figure 5: Formal definitions for local compilation to FDDs, from [29].

NetKAT treats a packet as a record of fields f ranging over standard headers such as Ethernet and IP source and destination, as well as logical fields such as `sw` and `pt`, which keep track of the switch and port where the packet is currently located in the network and are useful for program analysis. Atomic terms in the language are predicates on, or modifications to, packet fields. Each predicate behaves like a filter on packets—packets that do not match the boolean condition encoded in the predicate are dropped. Predicates include primitive tests on field values ($f = n$), as well as standard boolean operators ($+$, $\cdot$, and $\neg$). Modifications ($f \rightarrow n$) update the field f with the value n. The union operator ($p + p'$) copies the input packet, processes one copy using p and the other copy using p', and takes the union of the resulting sets of packets. Note that some operators are overloaded and can be applied to predicates and policies—e.g., $+$ is meant to represent disjunction on predicates and union on policies. The behavior specified in the denotational semantics in Figure 3 captures both cases. The sequential composition operator ($p \cdot p'$) processes the input packet using p and then feeds each output of p into p'. This form of composition is denoted by $\bullet$. Iteration p^* behaves like the union of p composed with itself zero or more times. To make authoring programs easier, links ($sw1 : pt1 \rightarrow sw2 : pt2$) and conditionals (if-then-else) are encoded as follows:

$$sw1 : pt1 \rightarrow sw2 : pt2 \triangleq$$
$$\quad sw = sw1 \cdot pt = pt1 \cdot sw := sw2 \cdot pt := pt2$$
$$\text{if } a \text{ then } p_1 \text{ else } p_2 \triangleq$$
$$\quad (a \cdot p_1) + (\neg a \cdot p_2)$$

We will use these constructs frequently in examples, although they do not increase the expressive power of the language.

4.1 Encoding Network-Wide Behavior

NetKAT can be used to specify both the forwarding functionality provided by the fabric, as well as the behavior required by a policy. The programmer specifies forwarding paths using predicates and modifications, combined using the regular operators described above. Since NetKAT expresses links as modifications to `switch` and `port` fields, a network topology can be encoded as a union of links. Furthermore, given predicates in and out defining ingress and egress locations, the end-to-end behavior of a network with forwarding policy p and topology t is described by:

$$in \cdot (p \cdot t)^* \cdot p \cdot out$$

Informally, this program accepts incoming packets and repeatedly processes them at switches and forwards them across links until they exit the network. We refer to this construct as the *analysis form* of a NetKAT program. A program in this form accepts incoming packets and repeatedly forwards them across switches and links in the topology, until they exit the network. Expressing network behavior in this form allows the compiler to perform analyses and generate data structures useful for our rewriting approach in Section 5. Finally from the analysis form, the compiler can generate forwarding tables for individual SDN switches. These flowtables together implement the forwarding behavior specified by the original NetKAT program [29, 13].

$$\mathcal{L}[\![\mathit{false}]\!] \triangleq \{\} \qquad \mathcal{L}[\![f{\leftarrow}n]\!] \triangleq \{\{f{\leftarrow}n\}\}$$
$$\mathcal{L}[\![\mathit{true}]\!] \triangleq \{\{\}\} \qquad \mathcal{L}[\![f{=}n]\!] \triangleq (f{=}n\,?\,\{\{\}\} : \{\})$$
$$\mathcal{L}[\![\neg p]\!] \triangleq \neg\mathcal{L}[\![p]\!] \qquad \mathcal{L}[\![p_1 + p_2]\!] \triangleq \mathcal{L}[\![p_1]\!] + \mathcal{L}[\![p_2]\!]$$
$$\mathcal{L}[\![p^*]\!] \triangleq \mathcal{L}[\![p]\!]* \qquad \mathcal{L}[\![p_1 \cdot p_2]\!] \triangleq \mathcal{L}[\![p_1]\!] \cdot \mathcal{L}[\![p_2]\!]$$

Figure 6: Local compilation to FDDs, from [29].

4.2 Circuit NetKAT

Although NetKAT allows the specification of global behaviors, it is sometimes less convenient for describing certain kinds of fabrics. In particular, circuit-based fabrics (such as optical networks) have their own set of constraints that are not enforced by default in NetKAT. For example, optical channels have the following con-

straints, which differentiate them from switches in packet networks:

1. **Optical continuity**: an incoming channel can be dropped or forwarded to an outgoing port, without changing the frequency slots the channel occupies.

2. **Split restriction**: an incoming channel cannot be forwarded to more than one outgoing ports.

3. **Merge restriction**: channels occupying overlapping frequency slots coming from multiple incoming ports cannot be merged to the same outgoing port.

4. **Transponder restriction**: a transponder port (that converts between electrical and optical signals) cannot input or output on more than one optical channel.

To enforce these constraints, we identify a constrained subset of NetKAT, called Circuit NetKAT. As its name suggests, a Circuit NetKAT program is a set of circuits, where each circuit is defined by a starting switch and port, a channel identifier, a list of hops and a final egress port, where each switch is an optical ROADM. The syntax is given in Figure 4a.

Circuit NetKAT programs are valid if they satisfy the validity conditions outlined in Figure 4b. Each circuit can be viewed as an *allocation (A)* from (switch, port) pairs to a channel identifier. The first condition (CONTINUITY) states that all points on the path defined by the circuit are allocated the same channel identifier. This satisfies the optical continuity restriction. The second condition (DISJOINTNESS) states that if two circuits use the same channel identifier, then their paths must be disjoint, i.e., at each port, a particular channel comes from, and is forwarded to, at most one destination. This condition satisfies the split, merge and transponder restrictions above.

A compiler takes a Circuit NetKAT program and checks if the program is valid according to the above conditions. If the program is valid, it is converted into a NetKAT program, otherwise it is rejected. This conversion simply involves inserting port assignments in between the links to properly forward signals from ingress to egress ports on each ROADM. The resulting program can then be compiled to flowtables that implement an optical forwarding fabric. In our implementation, we developed an extension to the NetKAT compiler backend so that the generated flowtables can be installed and tested on an optical network simulator (described in Section 6).

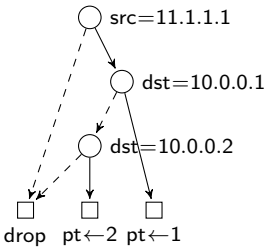
4.3 Forwarding Decision Diagrams

NetKAT programs can be compiled to an intermediate representation called a Forwarding Decision Diagram (FDD) [29]. FDDs are generalizations of structures called binary decision diagrams [1]. They are trees where internal nodes represent tests on packet headers, each with a "true" and a "false" branch. Leaf nodes are sets of modifications to packet headers. Figure 7 shows an example NetKAT program and the FDD it generates.

A leaf node in the FDD is a set of actions, denoted $\{a_1, \ldots, a_k\}$. An action a maps fields to values: $\{f_1 \leftarrow n_1, \ldots, f_k \leftarrow n_k\}$ with each field occuring at most once. An internal node, written as $(f{=}n\,?\,d_1 : d_2)$, is specified by a test $f{=}n$ and two sub-diagrams. If the packet satisfies the test the true branch (d_1) is evaluated, otherwise the false branch (d_2) is evaluated. FDDs must also satisfy well-formedness judgments ensuring that tests appear in a consistent order and do not contradict previous tests to the same field.

The NetKAT primitives *true*, *false*, and $f \leftarrow n$ all compile to simple leaf nodes. The empty action set $\{\}$ drops all packets while the singleton action set $\{\{\}\}$ contains the identity action $\{\}$ which

```
if src=11.1.1.1 then
 (if dst=10.0.0.1 then
   port:=1
  else if dst=10.0.0.2 then
   port:=2
  else
   false)
else
 false
```



α	β
¬src=11.1.1.1	drop
src=11.1.1.1 ∧ ¬dst=10.0.0.2 ∧ ¬dst=10.0.0.1	drop
src=11.1.1.1 ∧ dst=10.0.0.1	pt←1
src=11.1.1.1 ∧ dst=10.0.0.2 ∧ ¬dst=10.0.0.1	pt←2

Figure 7: Example NetKAT program, FDD, and dyads.

copies packets unchanged. NetKAT tests $f{=}n$ compile to a conditional whose branches are the FDDs for *true* and *false* respectively. The union operator $(d_1 + d_2)$ traverses d_1 and d_2 and takes the union of the action sets at the leaves. Sequential composition $(d_1 \cdot d_2)$ merges two packet-processing functions into a single function. It uses auxiliary operations $d\,|_{f=n}$ and $d\,|_{f\neq n}$ (described in Figure 5) to restrict a diagram d by a positive or negative test respectively. The FDD Kleene star operator $d*$ is defined using a fixed-point computation. The well-formedness conditions on FDDs ensure that such a fixed point exists. The compilation process is formalized in Figures 5 and 6 and is described in further detail in the original paper on the NetKAT compiler [29].

4.4 Generating Dyads from FDDs

A depth-first search over the FDD lets us collect up pairs of predicates (the conjunction of internal nodes, denoted α) and corresponding modifications (a leaf node, denoted β) that encode the input-output behavior of the program. That is, by compiling a NetKAT program to an FDD, we can easily produce a compact representation of its forwarding behavior that can be used for further analysis. We call each pair of a predicate α and its corresponding modifications β a *dyad*. Since the analysis form includes the topology and ingress and egress predicates, α includes the starting switch and port, and β includes the destination switch and port. Thus a dyad captures the source and sink of all traffic satisfying a given predicate.

Figure 7 shows a simple NetKAT program, the corresponding FDD and the generated dyads. There are four paths from the root node of the FDD to a leaf. The two leftmost paths lead to a drop node. The rightmost path checks the source IP (src=11.1.1.1) and destination IP (dst=10.0.0.1) and forwards out port 1. The remaining path checks the source and destination IP addresses, must also check for the negation of the previous destination IP address.

Performing this analysis on the programmer-supplied policy gives us the *required* sources and sinks of each traffic class. The same analysis on the fabric gives us sources and sinks of each path *provided* by fabric. The remainder of our system leverages the FDD structure and the dyads derived from them to determine how required functionality can be mapped onto an existing fabric. Section 5 discusses the requirements for a correct mapping and describes our architecture and approach.

5. COMPILATION TO THE EDGE

Next we describe how to implement policies expressed in NetKAT into an equivalent edge implementations. We first present a basic

compilation algorithm, and then discuss extensions with segmented paths and constraints on paths.

5.1 Problem Statement

Our goal is to start with a network policy and generate edge configurations that leverage the fabric for connectivity. Accordingly, our compiler takes as inputs a network policy and a forwarding fabric (both expressed as NetKAT programs), as well as a set of edge switches to target. Additionally, we assume knowledge of the physical topology, such as the ingress/egress predicates for both the policy and fabric. Given these inputs, we generate ingress and egress NetKAT programs, with the following properties:

1. **Edge implementation:** Both programs can be implemented entirely on edge switches—i.e., any `switch` predicates in the generated programs only match edge switches.

2. **Ingress classification:** The ingress program implements the same traffic classification as the user policy—i.e., the union of all the αs derived from the user policy's FDD.

3. **Egress modification:** The egress program implements the same modifications to packet header fields as the user policy—i.e., for each α implemented by the ingress program, the egress program must apply the corresponding β to the same traffic class.

4. **Fabric transit:** From each (α, β) pair derived from the policy's FDD, the fabric forwards from the source location in α to the sink location in β.

Together, these properties ensure a "one big switch" abstraction [18, 3]—a combination of edge program and fabric is equivalent to a policy program if they produce the same input/output behavior. Formally, if f and p are the NetKAT programs for the fabric and policy, then let $\phi_f = [\![f]\!], \phi_p = [\![p]\!]$ denote the corresponding packet forwarding functions according to the NetKAT semantics in Figure 3. The desired edge forwarding functions are given by ϕ_i (for ingress), and ϕ_o (for egress). The correctness condition for a compiler implementing the "one big switch" abstraction is captured by the following equivalence:

$$\phi_i \bullet \phi_f \bullet \phi_o \equiv \phi_p$$

Our compiler computes programs i and o such that $\phi_i = [\![i]\!]$ and $\phi_o = [\![i]\!]$, or fails if no such programs exist.

5.2 Basic Compiler

For both the fabric and user policy we produce an analysis form as described in Section 4.1. The NetKAT compiler generates Forwarding Decision Diagrams for each program. By iterating through the FDD, we convert each program to a set of dyads—pairs of predicates (α) and modifications (β). Since the analysis form includes the topology as well as network ingresses and egresses, αs include the starting switch and port (sources), and βs include the destination switch and port (sinks). In order to correctly implement a policy using an existing fabric, we need to match the sources and sinks required by the policy's dyads to those provided by the fabric. We've explored two approaches to solving this matching problem.

Our first approach is based on simple graph algorithms. We construct a *connectivity graph* G where the nodes are the sources and sinks of the user policy. There is an edge between two nodes if they are connected via a path in the fabric. We determine this by iterating through the (α, β) pairs for the fabric, and adding an edge to G if the α contains (or is one hop away from) a source and the corresponding β contains the sink (or is one hop away from it). To connect a source and sink, we simply check whether there is an edge between them in G.

	Element	Definition
Input	F	Fabric dyads, indexed by j
	P	Policy dyads, indexed by i
	src(d)	A function (Dyad $\rightarrow$ switch)
	dst(d)	A function (Dyad $\rightarrow$ switch)
Generated	$V_{i,j}$	Dyad i possibly implemented by dyad j
Output	$V_{i,j} = 1$	Dyad i implemented by dyad j

$$\text{Minimize} \quad \sum_{\substack{i \in \mathrm{P} \\ j \in \mathrm{F}}} V_{i,j}$$

$$\text{such that} \quad \forall i \in \mathrm{P} \qquad \sum_{j \in \mathrm{F}} V_{i,j} = 1$$

$$\forall i \in \mathrm{P}, \forall j \in \mathrm{F} \qquad V_{i,j} = 0 \text{ iff } \mathrm{src}(P_i) \neq \mathrm{src}(F_j) \vee \mathrm{dst}(P_i) \neq \mathrm{dst}(F_j)$$

Figure 8: Dyad matching as an LP problem.

Our second approach is based on a formulation as a linear programming problem whose solution is a matching between policy and fabric dyads. We generate a sequence of variables $V_{i,j}$ denoting the *possibility* of policy dyad i using the fabric dyad j. If the endpoints of policy dyad i and the fabric dyad j are not identical (or adjacent), then we generate a constraint limiting $V_{i,j} = 0$. In the basic case, we choose a single fabric dyad to implement each policy dyad. This is enforced by a constraint $\sum_{j \in \text{fabric}} V_{i,j} = 1$ for each policy dyad j. The full LP formulation is given in Figure 8.

Using an LP formulation is more powerful than is strictly needed, but it allows for better extensibility. To include additional features such as path constraints, we simply need to add more variables and constraints to the LP problem. Without it, we would have to write custom analyses over the dyads or the connectivity graph for the same functionality. Note also that the generic objective function used here could be replaced with network-specific objectives.

After a matching fabric dyad is found for each policy dyad, we need to generate ingress and egress programs that implement the policy using the fabric. We use each α in the policy as the predicate for a forwarding rule on the source switch, and generate output actions for the rule in two steps. At the source, since more than one stream of traffic may take the same path through the fabric, we attach a tag (eg, a VLAN tag) unique to this α to each packet. The collection of these forwarding rules form the required *ingress program.* Similarly, on the corresponding sink we install the matching β, modifying it to act only on traffic matching the tag attached by the source. These modified βs form the *egress program*.

5.3 Segmented Path Compilation

The "one big switch" abstraction allows network administrators to specify the endpoints of a particular class of traffic. A natural extension is to allow specifying an entire path, (e.g., $s_1 \Rightarrow s_2 \Rightarrow s_3$) instead of just a source-sink pair. Such a segmented path connects intermediate nodes that are part of the user-controlled edge. For each neighboring pair of nodes in the chain, the compiler would have to find a connecting segment through the fabric. The segments are then chained together to construct the whole path. Just as in dyad matching, we use a unique tag (e.g., a VLAN tag) to differentiate traffic classes and track them across segments. An ingress program that matches the policy's α and attaches the appropriate

Listing 5: Multi-segment program for applying a firewall.

```
if dst=backend and tcpDst=80 then
  frontend ==> firewall ==> backend
else if dst=backend and tcpDst=22 then
  frontend ==> backend
else if dst=frontend then
  backend ==> frontend
```

Listing 6: Ingress and egress program for firewall application.

```
if switch=2 and port=1 and dst=frontend then
  vlanId := 3; port := 2
else if switch=1 and port=1 and
      tcpDst=22 and dst=backend then
  vlanId := 2; port := 3
else if switch=1 and port=1 and
      tcpDst=80 and dst=backend then
  vlanId := 1; port := 2

if vlanId=3 and switch=1 and port=3 then
  strip vlan; port := 1
else if vlanId=2 and switch=3 and port=2 then
  strip vlan; port := 1
else if vlanId=1 and switch=2 and port=2 then
  strip vlan; port := 1
else if vlanId=1 and switch=3 and port=2 then
  strip vlan; port := 1
```

tag is installed at the start of the path. At each intermediate node, we send traffic from the fabric to the edge switch, and install *bounce* programs that examine the tag and return traffic to the fabric. Finally, we install an egress program at the end to match the tag and apply modifications according to the policy's β. Note that we only install rules on the relevant edge switches, without modifying the fabric connecting them. Thus the core of the network can remain static, reducing the overhead in changing network policy.

Segmented paths are simply an extended form of dyad matching. We find a fabric dyad to carry the traffic in between each consecutive pair of points. We extend our LP back end with some additional bookkeeping to reuse the same tag across each segment, and then apply the proper modifications at the end. NetKAT can already describe paths by specifying each hop, as shown in our motivating example in Listing 2. This extension allows us to to describe general paths while letting the compiler determine the specific hops. Such policies are useful for applications involving service chaining and middleboxes—e.g., network functions such as firewalls and intrusion detection are implemented on nodes at various points in the network, and simpler switches in the core of the network move traffic to the required processing nodes.

For example, Listing 5 shows a NetKAT program that directs Web requests from a front-end to a back-end through the firewall, but SSH traffic and traffic from back-end to front-end can pass directly through. An optical fabric similar to that in Figure 2 could support this program, with the firewall, front-end and backend replacing the hosts. The fabric program would be similar to Listing 2. Listing 6 shows generated ingress and egress programs. VLAN tags are used to separate different traffic classes. We assume that the intermediate nodes require the original traffic, without tags. Therefore tags are removed and reapplied at the end of every segment.

5.4 Compilation With Path Constraints

Segmented paths allow the policy to direct traffic across multiple points on the edge. However they do not provide any control over how the fabric is utilized—the compiler chooses any available fabric dyad with matching end-points. Finer-grained control can be exposed by incorporating path constraints—i.e., instead of allowing arbitrary intermediate nodes on the path that implements a policy, we allow the programmer to specify constraints on the paths that traffic must pass through. To find a matching fabric dyad, we need to consider the entire path represented by the dyad, not just the endpoints. Only the fabric dyads whose paths contain all the required nodes can be used to carry the policy traffic. This form of path constraint is particularly useful in the optical domain—optical signals need to be regenerated after being transmitted for a certain distance, but only certain nodes in an optical fabric are equipped with regenerators. By specifying that the appropriate nodes with regenerators must be visited as points on the path, a policy can ensure that traffic will reach its destination.

Starting from the linear programming formulation described in Section 5.2, we add more constraints to capture the fabric's provided paths and the policy's required intermediate nodes. First, we use the NetKAT compiler framework to produce a mapping from fabric dyads to paths (the $path(d)$ function in Figure 9). This can be done by symbolically executing the NetKAT program with respect to the given dyad. From the policy, we produce a mapping from policy dyads to the intermediate nodes required for each dyad. This is represented by the $pcs(d)$ function in Figure 9

We generate a variable $N_{n,j}$ for each dyad j, and node n in the fabric. We generate constraints setting $N_{n,j} = 1$ if n is on the path for dyad j and 0 otherwise. Recall that in the original LP formulation we generate variables $V_{i,j}$ representing the possibility of policy dyad i being implemented using fabric dyad j. We constrain $V_{i,j} = 0$ unless the dyad endpoints are the adjacent. If policy dyad i also specifies nodes $n_0 \ldots n_k$ as intermediates, we generate further constraints that set $V_{i,j} = 0$ unless all of $N_{n_0,j} \ldots N_{n_k,j}$ are 1. The extended LP formulation is shown in Figure 9.

	Element	Definition
Input	F	Fabric dyads, indexed by j
	P	Policy dyads, indexed by i
	src(d)	Function: Dyad $\rightarrow$ source switch
	dst(d)	Function: Dyad $\rightarrow$ sink switch
	path(d)	Function: Fabric dyad $\rightarrow$ nodes on path
	pcs(d)	Function: Policy dyad $\rightarrow$ path constraints
Generated	$V_{i,j}$	Dyad i possibly implemented by dyad j
	$N_{n,j}$	Fabric nodes n used by dyad j
Output	$V_{i,j} = 1$	Dyad i implemented by dyad j

$$\text{Minimize} \quad \sum_{\substack{i \in P \\ j \in \mathrm{F}}} V_{i,j}$$

such that

$$\forall i \in \mathrm{P} \quad \sum_{j \in \mathrm{F}} V_{i,j} = 1$$

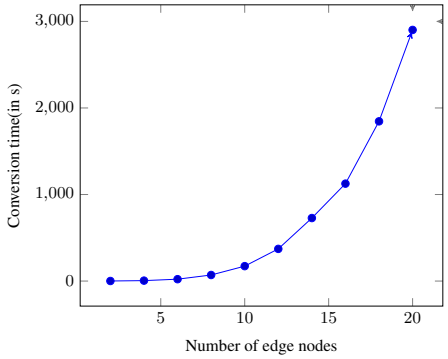
$$\forall n \in \text{nodes}, \forall j \in \mathrm{F} \quad N_{n,j} = 0 \quad \text{iff } n \notin \text{path}(j)$$

$$\forall i \in \mathrm{P}, \forall j \in \mathrm{F} \quad V_{i,j} = 0 \quad \text{iff} \quad \sum_{n \in pcs(i)} N_{n,j} < |pcs(i)|$$

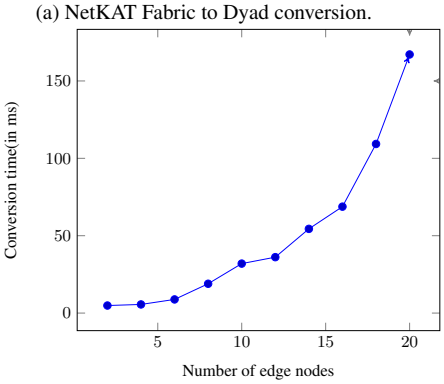
$$\forall i \in \mathrm{P}, \forall j \in \mathrm{F} \quad V_{i,j} = 0 \quad \text{iff} \quad \text{src}(P_i) \neq \text{src}(F_j) \vee \text{dst}(P_i) \neq \text{dst}(F_j)$$

Figure 9: Dyad matching with path constraints as an LP problem.

Figure 10: CORONET 60-node optical topology [4].



(a) NetKAT Fabric to Dyad conversion.



(b) NetKAT Policy to Dyad conversion.

Figure 11: Dyad conversion scalability.

6. IMPLEMENTATION

We have developed a prototype implementation of our edge compilation framework as well as a supporting simulation and testing system. The dyad matching algorithm, segmented paths and path constraint extensions are implemented atop the NetKAT compiler. The iteration over FDDs to produce dyads is implemented as a 400-line OCaml module. A 300-line OCaml module serializes the linear programming problems to a format understood by the Gurobi solver. This serialization module is used by both the dyad matching back-end (100 lines of OCaml) and the path constraints back-end (115 lines). Syntax extensions to support Circuit NetKAT and segmented paths are another 300 lines of OCaml.

We also develop a simulator for hybrid packet-optical networks as in Section 2. The open-source Linc-OE software switch simulates ROADMs, and supports an API based on the OpenFlow 1.3 protocol to set up the optical channels. The packet switches are simulated using the Mininet simulator [22], and we developed a 200-line Python extension to embed Linc-OE switches in Mininet. To control the switches in the network, we developed a packet-optical OpenFlow controller in Java using the OpenFlowJ library (500 lines of Java). The controller accepts switch forwarding tables as emitted by compiler and installs them on the simulated switches. The examples in Sections 2 and 5.3 have been tested using this simulator.

Finally, we developed a set of tools to generate optical fabrics and test policies based on network topologies. We use these tools to perform scalability tests on our system using a real-world topology, as described in the next section.

7. EVALUATION

We evaluate our compiler on CORONET (Figure 10), a real-world optical topology with 60 nodes that is representative of current carrier networks [4]. The network stretches across the continental United States with three link-diverse cross-continental paths. We use this topology to generate realistic optical fabrics and policies and measure our compiler's performance on a variety of inputs. All experiments were run on Dell r620 servers, equipped with two eight-core 2.60 GHz E5-2650 Xeon CPUs and 64 GB of RAM running Ubuntu 14.04.1 LTS.

7.1 Topologies, Fabrics, and Policies

To generate the optical topology, we place a ROADM at each node in the physical topology. Then we attach packet switches to the ROADMs on the two coasts, using a configurable number of transponder ports. The packet switches constitute a flexible edge network to use as the target for our generated edge programs. Given this topology, we generate a range of fabrics and edge policies.

To generate the fabric, we choose a subset of ROADM nodes located along either coast. From this subset, we randomly choose pairs of nodes (R_e, R_w), such that each is on an opposite coast. We then find three paths between R_e and R_w such that each path goes over a different physically disjoint cross-continental path. For each such path, we create a unique optical channel between R_e and R_w. Using these channels, we generate a NetKAT program to implement the fabric. The program matches traffic incoming on the transponder ports (connected to the packet switches) and places them on an appropriate channel. The channels are forwarded across the network using the appropriate paths. At the egress, the program matches optical channels and outputs them to a transponder port. By increasing the number of coastal ROADMs that we connect via optical channels we can scale the size of the fabric. Using this fabric, we can generate policies to test the various parts of our system.

To test the scalability of the basic dyad matching functionality, we generate policies to provide cross-country connectivity. We start with pairs of nodes (P_e, P_w) such that each is a packet switch on opposite coasts. We use NetKAT predicates to separate out and forward a number of traffic classes between each P_e and P_w. By connecting increasing number of edge nodes, we can increase the size and complexity of the policy.

7.2 Dyad Generation Scalability

The first step in producing the required edge programs is to convert the fabric and the policy from NetKAT programs into dyads for the later stages. This computation is the same irrespective of how the dyads are matched. Figure 11(a) shows the time taken to convert the fabric into dyads using the NetKAT compiler framework. Figure 11(b) shows the time taken to convert the policy into dyads.

The time taken to produce the dyads for the fabric is the largest—nearly an hour for the largest fabrics. This time dominates the conversion time for the policy (less than 200 ms). However, one of our motivating constraints is that the fabric is rigid and changes infrequently, while the policy may change often. Thus the dyad conversion for the fabric could easily be performed just once and then cached for subsequent changes to the policy. Moreover, we believe its performance can be further improved by adding further optimizations to the NetKAT compiler.

7.3 Dyad Matching Scalability

The results of running our dyad-matching back-end on increasing fabric and policy sizes is shown in Figure 12. Each graph plots the number of coastal nodes on the X-axis, and the time taken to complete each stage of the matching process on the Y-axis. Figures 12(a-c) show the time taken to generate the matching problem, the time to solve the problem, and the time to generate the NetKAT edge programs to implement the found matching respectively.

The graphical matching approach completes in microseconds, while generating the LP problem and solving it takes only hundreds of milliseconds. In either case, even very complex changes to the policy across the entire topology can be handled in a small amount of time. The time penalty paid for the LP approach is made up for by flexibility—generating the extra linear constraints for implementing path constraints takes only 10 lines of OCaml.

7.4 Path Constraint Scalability

We can now extend the basic matching evaluation with path constraints. Starting from the same fabric and policy setup, we add an increasing number of intermediate nodes to the policy. For each bicoastal pair of nodes (P_e, P_w) we add an increasing number of intermediate points drawn from one of the physical paths connecting them. Figures 13(a-c) show the time taken to generate the matching problem, the time to solve the problem, and the time to generate the NetKAT edge programs to implement the found matching respectively. The baseline case (0 intermediate nodes) is the dyad matching approach described in the previous section.

Using a linear programming solver allows us to support path constraints by simply adding more variables and constraints. However, adding this functionality considerably increases the size of the LP problems, increasing the time taken to generate, solve and recover the solution. Though the time penalty is significant in going from the basic to the smallest number of constraints, there are no additional penalty to increasing the number of intermediate nodes. This is to be expected, since the majority of the LP problem is composed of variables and constraints representing the fabric paths, rather than the policy's intermediate nodes.

7.5 Discussion

The process for generating edge programs from NetKAT fabric and policy programs can be broken down into four stages—preprocessing the programs into dyads, formulating the matching problem, solving the matching problem and finally generating edge programs from the solutions. The preprocessing step converts the programs into their dyad representation. Applying this step to the fabric takes the longest amount of time (on the order of multiple minutes to an hour). However, since our use cases are networks where the core fabric is meant to be rigid and change rarely, this step would only be performed infrequently. Formulation and solution takes on the order of milliseconds for the basic case, and less than a second if we include path constraints. Generating the final edge programs takes even less time.

These experiments suggest that our framework could be used to generate edge programs from network policies, provided the fabric is stable for a relatively long period of time. This is a reasonable assumption for both the optical circuit networks we've been considering, as well as other fabrics involving legacy, non-SDN devices [23, 7]. Thus we believe that our system provides a practical method for flexible network management.

8. RELATED WORK

Our approach leverages a number of theoretical advances made in previous work on NetKAT [2]. In addition we rely on the existing compiler infrastructure for NetKAT to produce the FDDs that our analysis takes as input [29].

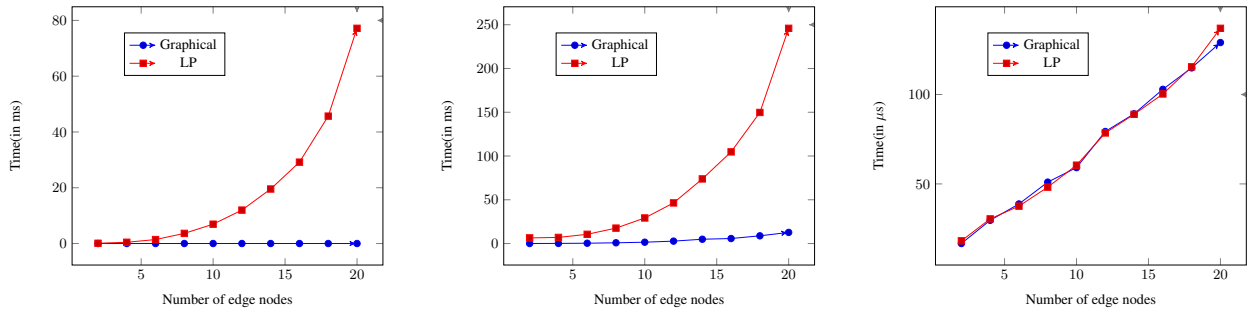
Optical networks. The flexibility of SDNs is attractive for optical networks as well. Recent work has identified a set of challenges that are different from that in packet networks [12]. In particular, signal attenuation at long distances is a significant problem that requires the careful placement of regenerator nodes. Our path constraints extension finds paths connecting the required regenerator, while previous work has tackled the problem of where in a network to place these regenerators in the first place [19].

A number of recent efforts have developed architectures and control abstractions for hybrid packet-optical networks. REACToR incorporates optical circuits into data center network with the goal of improving performance without sacrificing control [24]. Another system, OWAN jointly manages the optical and packet layers to optimize bulk data transfers in wide-area networks [17]. Our work, which focuses on mechanisms for implementing programs at the edge, is complementary to these efforts.

Edge-fabric distinction. Several existing systems are also based on an edge-fabric distinction. An early paper by Casado et al. [7] proposed an architecture based on a fabric service model. The same paper discussed the problem of mapping policies to the edge but did not propose a solution. Panopticon addresses incremental deployment of SDNs using data structures called Solitary Confinement Trees, which are similar to our fabrics [23]. However, Panopticon focuses on L2/L3 packet networks in small to medium scale enterprises, rather physically heterogenous deployments. As mentioned above, Felix is also based on an edge-fabric distinction and adopts a similar approach based on NetKAT and FDDs [8].

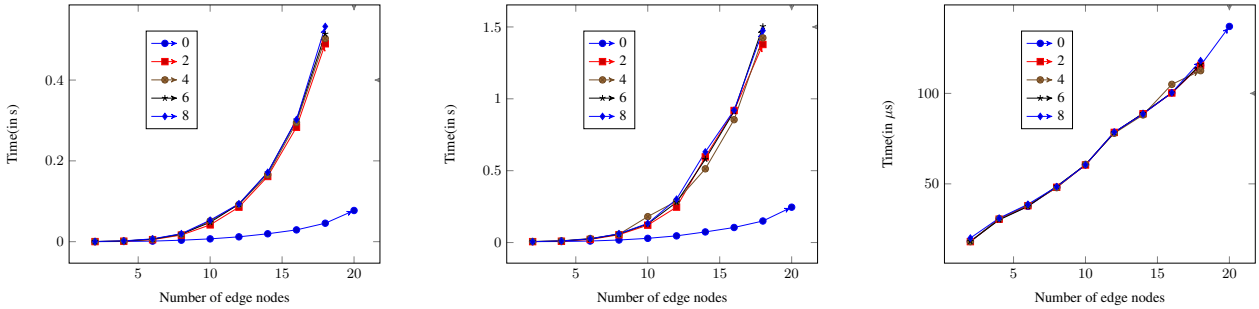
SDN controller frameworks. ONIX was an influential early SDN controller that offered a number of features including slicing and virtualization [21]. These features are also realized in more recent work on VMware NSX [20]. Exodus translates SDN policies into configurations that can be installed on legacy devices [28]. Finally, Fibbing developed approaches for implementing SDN-like control using distributed routing protocols [32].

Software synthesis for networks. Several recent systems have applied ideas from program synthesis to networks. Software synthesis is attractive because it offers the promise of finding general solutions to a wide class of problems, rather than relying on ad-hoc and possibly brittle solutions to particular problems. Genesis and SyNet synthesize device-level configurations from high-level policies that incorporate path and traffic engineering constraints [31, 9]. Our approach is also based on synthesis, but we exploit domain-specific knowledge to produce dyads by analyzing NetKAT programs, that are then fed to a solver to compute a matching. By doing so, we cut down the space of possible solutions that a solver has to investigate.



(a) Time to create matching problem. (b) Time to solve matching problem. (c) Time to generate edge programs.

Figure 12: Scalability of LP and graphical dyad matching.



(a) Time to create matching problem. (b) Time to solve matching problem. (c) Time to generate edge programs.

Figure 13: Scalability of matching with path constraints.

Network programming languages. There is a also large body of work on domain-specific programming languages for SDN. Examples include Frenetic [10], Nettle [33], Pyretic [26], NetCore [25], NetKAT [2], Maple [34], and FlowLog [27], among others. The compilers for these languages translate high-level programs into switch-level forwarding rules—i.e., they assume that the network consists of SDN switches that can be frequently reprogrammed in response to changing conditions. An important difference is that these systems employ simple forms of tagging and encapsulation whereas our system leverages whatever mechanism the underlying fabric provides.

9. FUTURE WORK

This paper focuses on compiling policies to edge programs, such that they make use of an existing fabric. We view this work as a first step toward exploring the interaction of flexible edges and rigid fabrics. We have explored some extensions to the basic "one big switch" programming model—segmented paths and path constraints. There are more interesting extensions that could be supported, including fault tolerance, load balancing and more interesting path constraints such as node or edge disjointness. Implementing these extensions will undoubtedly require further development of the linear programming based back-end, or exploring techniques based on counter-example guided inductive synthesis (CEGIS) [30].

Another line of future work involves generating more than just the NetKAT edge programs. There may be cases where an existing fabric cannot be used to implement a given policy. In such cases, the compiler could suggest minimal extensions to the fabric required to support the policy, or conversely, precisely locate portions of the policy that the fabric cannot support. Again, CEGIS seems like a promising approach for such extensions.

10. CONCLUSION

This paper tackles the problem of deploying high-level user policies to heterogenous networks. We assume a rigid core network, surrounded by flexible, SDN-enabled edge switches. Leveraging the NetKAT compiler framework we develop techniques to compile user policies into programs that only modify configurations on edge switches, using the fabric to provide connectivity. We focus on a particular class of such fabrics—optical circuit networks—and develop language extensions and testbeds to properly utilize them. Our system provides a practical way to reap the benefits of high-level network programming languages, while operating on the heterogenous networks deployed today. In the future, we intend to explore useful extensions to the policies, possibly using techniques such as counter-example guided inductive synthesis.

Acknowledgments. The authors wish to thank the ANCS reviewers for helpful feedback. Our work is supported by the NSF under grants CNS-1111698, CNS-1413972, CCF-1422046, CCF-1253165, and CCF-1535952; the ONR under grant N00014-15-1-2177; and gifts from Cisco, Facebook, Google, and Fujitsu.

11. REFERENCES

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 113–126, January 2014.

[3] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 29–43, August 2016.

[4] M. N. Architects. Coronet optical topology. Available at http://www.monarchna.com/topology.html, October 2006.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 1–12, August 2007.

[7] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Proceedings of the third workshop on Hot topics in software defined networking, HotSDN '12*, pages 85–90, August 2012.

[8] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang. Felix: Implementing traffic measurement on end hoses using program analysis. In *ACM SIGCOMM Symposium on Software-Defined Networking Research (SOSR)*, March 2016.

[9] A. El-Hassany, P. Tsankov, L. Vanbever, and M. T. Vechev. Network-wide configuration synthesis. *CoRR*, abs/1611.02537, 2016.

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011*, pages 279–291, 2011.

[11] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for NetKAT. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, pages 343–355, January 2015.

[12] S. Gringeri, N. Bitar, and T. J. Xia. Extending software defined network principles to include optical transport. *IEEE Communications Magazine*, 51(3):32–40, March 2013.

[13] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 483–494, 2013.

[14] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM 2013 Conference, SIGCOMM'13*, pages 15–26, August 2013.

[15] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally Deployed Software Defined WAN. In *ACM SIGCOMM 2013 Conference, SIGCOMM'13*, pages 3–14, August 2013.

[16] X. Jin, J. Gossels, J. Rexford, and D. Walker. CoVisor: A compositional hypervisor for software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15*, pages 87–101, May 2015.

[17] X. Jin, Y. Li, D. Wei, S. Li, J. Gao, L. Xu, G. Li, W. Xu, and J. Rexford. Optimizing bulk transfers with software-defined optical WAN. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 87–100, August 2016.

[18] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *Conference on emerging Networking Experiments and Technologies, CoNEXT '13*, pages 13–24, December 2013.

[19] I. Kim, P. Palacharla, X. Wang, Q. Zhang, D. Bihon, M. D. Feuer, and S. L. Woodward. Regenerator predeployment in cn-roadm networks with shared mesh restoration. *J. Opt. Commun. Netw.*, 5(10):A213–A219, Oct 2013.

[20] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, J. G. Igor Ganichev, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, , R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014*, pages 203–216, April 2014.

[21] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010*, pages 351–364, October 2010.

[22] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[23] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *2014 USENIX Annual Technical Conference, USENIX ATC '14*, pages 333–345. USENIX Association, June 2014.

[24] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, and G. Porter. Circuit switching under the radar with REACToR. In *NSDI*, pages 1–15, 2014.

[25] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pages 217–230. ACM, January 2012.

[26] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013*, pages 1–13. USENIX Association, April 2013.

[27] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014*, pages 519–531. USENIX Association, April 2014.

[28] T. Nelson, A. D. Ferguson, D. Yu, R. Fonseca, and S. Krishnamurthi. Exodus: Toward automatic migration of enterprise network configurations to sdns. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15*, pages 13:1–13:7. ACM, June 2015.

[29] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha. A fast compiler for NetKAT. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 328–341. ACM, September 2015.

[30] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.*, 40(5):404–415, October 2006.

[31] K. Subramanian, L. D'Antoni, and A. Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *POPL*, pages 572–585, 2017.

[32] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central Control Over Distributed Routing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015*, pages 43–56. ACM, August 2015.

[33] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In R. Rocha and J. Launchbury, editors, *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011*, volume 6539 of *Lecture Notes in Computer Science*, pages 235–249. Springer, January 2011.

[34] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM 2013 Conference, SIGCOMM'13*, pages 87–98. ACM, August 2013.