WNetKAT: A Weighted SDN Programming and Verification Language[*]

Kim G. Larsen[1], Stefan Schmid[2], and Bingtian Xue[3]

1 **Aalborg University, Aalborg, Denmark**
kgl@cs.aau.dk
2 **Aalborg University, Aalborg, Denmark**
schmiste@cs.aau.dk
3 **Aalborg University, Aalborg, Denmark**
bingt@cs.aau.dk

Abstract

Programmability and verifiability lie at the heart of the software-defined networking paradigm. While OpenFlow and its match-action concept provide primitive operations to manipulate hardware configurations, over the last years, several more expressive network programming languages have been developed. This paper presents *WNetKAT*, the first network programming language accounting for the fact that networks are inherently weighted, and communications subject to capacity constraints (e.g., in terms of bandwidth) and costs (e.g., latency or monetary costs). *WNetKAT* is based on a syntactic and semantic extension of the NetKAT algebra. We demonstrate several relevant applications for *WNetKAT*, including cost- and capacity-aware reachability, as well as quality-of-service and fairness aspects. These applications do not only apply to classic, splittable and unsplittable (s,t)-flows, but also generalize to more complex (and stateful) network functions and service chains. For example, *WNetKAT* allows to model flows which need to traverse certain waypoint functions, which can change the traffic rate. This paper also shows the relationship between the equivalence problem of WNetKAT and the equivalence problem of the weighted finite automata, which implies undecidability of the former. However, this paper also shows the decidability of whether an expression equals to 0, which is sufficient in many practical scenarios, and we initiate the discussion of decidable subsets of the whole language.

1998 ACM Subject Classification C.2 Computer-Communication Networks

Keywords and phrases Software-Defined Networking, Verification, Reachability, Stateful Processing, Service Chains, Weighted Automata, Decidability, NetKAT

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.18

1 Introduction

Managing and operating traditional computer networks is known to be a challenging, manual and error-prone process. Given the critical role computer networks play today, not only in the context of the wide-area Internet but also of enterprise and data center networks, this is worrisome. Software-Defined Networks (SDNs) in general and the OpenFlow standard in particular, promise to overcome these problems by enabling automation, formal reasoning and verification, as well as by defining open standards for vendors. Indeed, there is also a wide consensus that formal verifiability is one of the key advantages of SDN over past attempts

* This work was partially supported by the ERC Advanced Grant *LASSO*, the Danish VILLUM project *ReNet* and the Sino-Danish Basic Research Center project *IDEA4CPS*.

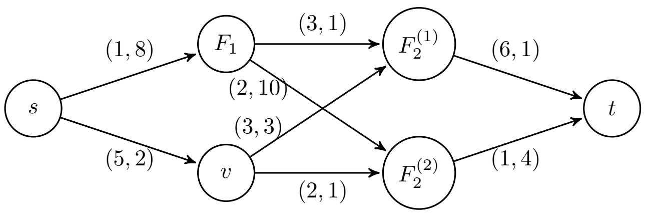

Figure 1 *Example:* A network hosting two (virtualized) functions F_1 and F_2. Function F_2 is allocated twice. The functions F_1 and F_2 may change the traffic rate.

to innovate computer networks, e.g., in the context of active networking [38]. Accordingly, SDN/OpenFlow is seen as a promising paradigm toward more dependable computer networks.

At the core of the software-defined networking paradigm lies the desire to program the network. In a nutshell, in an SDN, a general-purpose computer manages a set of programmable switches, by installing rules (e.g., for forwarding) and reacting to events (e.g., newly arriving flows or link failures). In particular, OpenFlow follows a match-action paradigm: the controller installs rules which define, using a *match* pattern (expressed over the *packet header fields*, and defining a *flow*), which packets (of a flow) are subject to which *actions* (e.g., forwarding to a certain port).

While the OpenFlow API is simple and allows to manipulate hardware configurations in flexible ways, it is very low level and not well-suited as a language for human programmers. Accordingly, over the last years, several more high-level and expressive domain-specific SDN languages have been developed, especially within the Frenetic project [13]. These languages can also be used to express fundamental network queries, for example related to *reachability*: They help administrators answer questions such as *"Can a given host A reach host B?"* or *"Is traffic between hosts A and B isolated from traffic between hosts C and D?"*.

What is missing today however is a domain-specific language which allows to describe the important *weighted aspects* of networking. E.g., real networks naturally come with capacity constraints, and especially in the Wide-Area Network (WAN) as well as in data centers, bandwidth is a precious resource. Similarly, networks come with latency and/or monetary costs: transmitting a packet over a wide-area link, or over a highly utilized link, may entail a non-trivial latency, and inter-ISP links may also be attributed with monetary costs.

Weights may not be limited to links only, but also nodes (switches or routers) have capacities and costs e.g., related to the packet rate. What is more, today's computer networks provide a wide spectrum of in-network functions related to security (e.g., firewalls) and performance (e.g., caches, WAN optimizers). To give an example, today, the number of so-called *middleboxes* in enterprise networks can be in the same order of magnitude as the number of routers [36]. A domain specific language for SDNs should be expressive enough to account for middleboxes which can change (e.g., compress or increase) the rate of the traffic passing through them. Moreover, a network language should be able to define that traffic must pass through these middleboxes in the first place, i.e., that routing policies fulfill waypointing invariants [39]. With the advent of more virtualized middleboxes, and the *Network Function Virtualization* paradigm, short *NFV*, (virtualized) middleboxes may also be *composed* to form more complex network services. For example, SDN traffic engineering flexibilities can be used to steer traffic through a series of middleboxes, concatenating the individual functions into so-called *service chains* [17, 26]. For instance, a network operator might want to ensure that all traffic from s to t should first be routed through a firewall FW, and then through a WAN optimizer WO, before eventually reaching t: the operator can do so by defining a service chain (s, FW, WO, t).

Let us consider a more detailed example, see the network in Figure 1: The network hosts two types of (virtualized) functions F_1 and F_2: possible network functions may include, e.g., a firewall, a NAT, a proxy, a tunnel endpoint, a WAN optimizer (and its counterpart), a header decompressor, etc. In this example, function F_2 is instantiated at two locations. Functions F_1 and F_2 may not be flow-preserving, but may *decrease* the traffic rate (e.g., in case of a proxy, WAN optimizer, etc.) or *increase* it: e.g., a tunnel entry-point may add an extra header, a security box may add a watermark to the packet, the counterpart of the WAN optimizer may decompress the packet, etc. Links come with a certain cost (say latency) and a certain capacity (in terms of bandwidth). Accordingly, we may annotate links with two weights: the tuple $(2, 3)$ denotes that the link cost is 2 and the link capacity 3. We would like to be able to ask questions such as: *Can source s emit traffic into the service chain at rate x without overloading the network?* or *Can we embed a service chain of cost (e.g., end-to-end latency) at most x?*.

1.1 Contributions

This paper initiates the study of weighted network languages for programming and reasoning about SDN networks, which go beyond topological aspects but account for actual resource availabilities, capacities, costs, or even stateful operations. In particular, we present *WNetKAT*, an extension of the *NetKAT* [1] algebra.

For example, *WNetKAT* supports a natural generalization of the reachibility concepts used in classic network programming languages, such as *cost-aware* or *capacity-aware reachability*. In particular, *WNetKAT* allows to answer questions of the form: *Can host A reach host B at cost/bandwidth/latency x?*

We demonstrate applications of *WNetKAT* for a number of practical use cases related to performance, quality-of-service, fairness, and costs. These applications are not only useful in the context of both splittable and unsplittable routing models, where flows need to travel from a source s to a destination t, but also in the context of more complex models with waypointing requirements (e.g., service chains).

The weighted extension of NetKAT is non-trivial, as capacity constraints introduce dependencies between flows, and arithmetic operations such as *addition* (e.g., in case of latency) or *minimum* (e.g., in case of bandwidth to compute the end-to-end delay) have to be supported along the paths. Therefore, we extend the syntax of NetKAT toward weighted packet- and switch-variables, as well as queues, and provide a semantics accordingly. In particular, one contribution of our work is to show for which weighted aspects and use cases which language extensions are required.

We also show the relation between WNetKAT expressions and weighted finite automata [9] – an important operational model for weighted programs. This leads to the undeciability of WNetKAT equivalence problem. However, leveraging this relation we also succeed to prove the decidability of whether an expression equals to 0: for many practical scenarios a sufficient and relevant solution. Moreover, this paper initiates the discussion of identifying decidable subsets of the whole language.

1.2 Related Work

Most modern domain-specific SDN languages enable automated tools for verifying network properties [12, 13, 29, 43, 44]. Especially reachability properties, which are also the focus in our paper, have been studied intensively in the literature [19, 20]. Indeed, the formal verifiability of the OpenFlow match-action interface [19, 20, 30, 46] constitutes a key advantage of the paradigm over previous innovation efforts [5]. Existing expressive languages use SAT formulas [27], graph-based representations [19, 20], or higher-order logic [45] to describe network topologies and policies.

Our work builds upon NetKAT, a new framework based on Kleene algebra with tests for specifying, programming, and reasoning about networks and policies. NetKAT respresents a more principled approach compared to prior work, and is also motivated by the observation that end-to-end functionality is determined not only by the behavior of the switches and but also by the structure of the network topology. NetKAT in turn is based on earlier efforts performed in the context of NetCore [28], Pyretic [29] and Frenetic [13]. It has recently been extended to a probabilistic [14] and temporal [2] setting, and first versions for specific use cases like QoS are emerging [34]. The Kleene algebra with tests was developed by Kozen [24]. However, to the best of our knowledge, there is no prior work on a general weighted (and stateful) version of NetKAT.

In general, stateful network design and analysis is very active field of research, and there are several interesting recent results, e.g., on the complexity and scalability of more stateful verification [42], on quantitative analysis [18], or on reachability [10].

2 Background

A Software-Defined Network (SDN) outsources and consolidates the control over data plane elements to a logically centralized control plane implemented in software. Arguably, software-defined networking in general, and its de facto standard, OpenFlow, are about programmability, verifiability and generality [11]: The behavior of an OpenFlow switch is defined by its configuration: a list of prioritized *(flow) rules* stored in the switch flow table, which are used to classify, filter, and modify packets based on their *header fields*. In particular, OpenFlow follows a simple match-action paradigm: the match parts of the flow rules (expressed over the header fields) specify which packets belong to a certain flow (e.g., depending on the IP destination address), and the action parts define how these packets should be processed (e.g., forward to a certain port). OpenFlow supports a rather general packet processing: it allows to match and process packets based on their Layer-2 (e.g., MAC addresses), Layer-3 (e.g., IP addresses), and Layer-4 header fields (e.g., TCP ports), or even in a protocol-independent manner, using arbitrary bitmasking [4].

OpenFlow also readily supports quantitative aspects, e.g., the selection of queues annotated with different round robin weights (the standard approach to implement quality-of-service guarantees in networks today), or meters (measuring the bandwidth of a flow). Moreover, we currently witness a trend toward more flexible and stateful programmable switches and packet processors, featuring group tables, counters, and beyond [3].

The formal framework developed in this paper is based on NetKAT [1]. NetKAT is a high-level algebra for reasoning about network programs. It is based on *Kleene Algebra with Tests (KAT)*, and uses an equational theory combining the axioms of KAT and network-specific axioms that describe transformations on packets (as performed by OpenFlow switch rules). These axioms facilitate reasoning about local switch processing functionality (needed during compilation and for optimization) as well as global network behavior (needed to check reachability and traffic isolation properties). Basically, an atomic NetKAT policy (a function from packet headers to sets of packet headers: essentially the per-switch OpenFlow rules discussed above) can be used to filter or modify packets. Policy combinators ($+$) allow to build larger policies out of smaller policies. There is also a sequential composition combinator to apply functions consecutively.

Besides the *policy*, modeling the per-switch OpenFlow rules, a network programming language needs to be able to describe the network *topology*. NetKAT models the network topology as a directed graph: nodes (hosts, routers, switches) are connected via edges (links) using (switch) *ports*. NetKAT simply describes the topology as the union of smaller policies that encode the behavior of each link. To model the effect of sending a packet across a link, NetKAT employs the sequential composition of a filter that retains packets located at one end of the link, and a modification that updates the switch and port fields to the location at the other end of the link. Note that the NetKAT topology and the NetKAT policy are hence to be seen as two independent concepts. Succinctly:

A *Kleene algebra* (KA) is any structure $(K, +, \cdot, ^*, 0, 1)$, where K is a set, $+$ and $\cdot$ are binary operations on K, * is a unary operation on K, and 0 and 1 are constants, satisfying the following axioms, where we define $p \le q$ iff $p + q = q$.

$$\begin{array}{ll}
p + (q + r) = (p + q) + r & p(qr) = (pq)r \\
p + q = q + p & 1 \cdot p = p \cdot 1 = p \\
p + 0 = p + p = p & p \cdot 0 = 0 \cdot p = 0 \\
p(q + r) = pq + pr & (p + q)r = pr + qr \\
1 + pp^* \le p^* & q + px \le x \Rightarrow p^*q \le x \\
1 + p^*p \le p^* & q + xp \le x \Rightarrow qp^* \le x
\end{array}$$

A *Kleene algebra with tests* (KAT) is a two-sorted structure $(K, B, +, \cdot, ^*, \bar{\ }, 0, 1)$, where $B \subseteq K$ and

- $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra;
- $(B, +, \cdot, \bar{\ }, 0, 1)$ is a Boolean algebra;
- $(B, +, \cdot, 0, 1)$ is a subalgebra of $(K, +, \cdot, 0, 1)$.

The elements of B are called *tests*. The axioms of Boolean algebra are:

$$\begin{array}{ll}
a + bc = (a + b)(a + c) & ab = ba \\
a + 1 = 1 & a + \bar{a} = 1 \\
a\bar{a} = 0 & aa = a
\end{array}$$

NetKAT is a version of KAT in which the atoms (elements in K) are defined over header fields f (variables) and values ω:

- $f \leftarrow \omega$ ("assign a value ω to header field f")
- $f = \omega$ ("test the value of a header field")
- dup ("duplicate the packet")

The set of all possible values of f is denoted Ω. For readability, we use *skip* and *drop* to denote 1 and 0, respectively.

The NetKAT axioms consist of the following equations, in addition to the KAT axioms on the commutativity and redundancy of different actions and tests, and enforcing that the field has exactly one value:

$$\begin{array}{rcll}
f_1 \leftarrow \omega_1; f_2 \leftarrow \omega_2 & = & f_2 \leftarrow \omega_2; f_1 \leftarrow \omega_1 \quad (f_1 \neq f_2) & (1) \\
f_1 \leftarrow \omega_1; f_2 = \omega_2 & = & f_2 = \omega_2; f_1 \leftarrow \omega_1 \quad (f_1 \neq f_2) & (2) \\
f = \omega; \mathsf{dup} & = & \mathsf{dup}; f = \omega & (3) \\
f \leftarrow \omega; f = \omega & = & f \leftarrow \omega & (4) \\
f = \omega; f \leftarrow \omega & = & f = \omega & (5) \\
f \leftarrow \omega_1; f \leftarrow \omega_2 & = & f \leftarrow \omega_2 & (6) \\
f = \omega_1; f = \omega_2 & = & 0 \qquad (\omega_1 \neq \omega_2) & (7) \\
\sum_{\omega \in \Omega} f = \omega & = & 1 & (8)
\end{array}$$

In terms of semantics, NetKAT uses *packet histories* to record the state of each packet on its path from switch to switch through the network. The notation $\langle pk_1, \dots, pk_n\rangle$ is used to describe a history with elements $pk_1, \dots, pk_n$ being packets; $pk :: \langle\rangle$ is used to denote a history with one element and $pk :: h$ to denote the history constructed by prepending pk on to h. By convention, the first element of a history is the current packet (the "head"). A NetKAT expression denotes a function $[\![\ ]\!] : H \to 2^H$, where H is the set of packet histories. Histories are only needed for reasoning: Policies only inspect or modify the first (current) packet in the history. Succinctly:

$$\begin{array}{rcl}
[\![f \leftarrow \omega]\!](pk :: h) & = & \{pk[\omega/f] :: h\} \\
[\![f = \omega]\!](pk :: h) & = & \begin{cases} \{pk :: h\} & \text{if } pk(f) = \omega \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\mathsf{dup}]\!](pk :: h) & = & \{pk :: pk :: h\} \\
[\![p + q]\!](h) & = & [\![p]\!](h) \cup [\![q]\!](h) \\
[\![pq]\!](h) & = & \bigcup_{h' \in [\![p]\!](h)} [\![q]\!](h') \\
[\![p^*]\!](h) & = & \bigcup_n [\![p^n]\!](h) \\
[\![0]\!](h) & = & \emptyset \\
[\![1]\!](h) & = & \{h\} \\
[\![\overline{a}]\!](h) & = & \begin{cases} \{h\} & \text{if } [\![a]\!](h) = \emptyset \\ \emptyset & \text{if } [\![a]\!](h) = \{h\} \end{cases}
\end{array}$$

▶ **Example 1.** Consider the network in Figure 1. NetKAT can be used to specify the topology as follows, where the field sw stores the current location (switch) of the packet:

$$\begin{array}{rcl}
t & ::= & sw = s; (sw \leftarrow F_1 + sw \leftarrow v) \\
 & & +sw = F_1; (sw \leftarrow F_2^{(1)} + sw \leftarrow F_2^{(2)}) \\
 & & +sw = v; (sw \leftarrow F_1^{(1)} + sw \leftarrow F_2^{(2)}) \\
 & & +sw = F_2^{(1)}; sw \leftarrow t \\
 & & +sw = F_2^{(2)}; sw \leftarrow t
\end{array}$$

The first line of the above NetKAT expression specifies that if the packet is at s, then it will be sent to F_1 or v. Analogously for the other cases. In OpenFlow, this policy can be implemented using OpenFlow rules, whose match part applies to packets arriving at s, and whose action part assigns the packets to the respective forwarding ports.

However, one can observe that with NetKAT it is not possible to specify or reason about the important quantitative aspects in Figure 1, e.g., the cost and capacity along the links or the function of F_2 which changes the rate of the flow. To do these, a weighted extension of NetKAT is needed.

3 WNetKAT

On a high level, a computer network can be described as a set of nodes (hosts or routers) which are interconnected by a set of links, hence defining the network topology. While this high-level view is sufficient for many purposes, for example for reasoning about reachability, in practice, the situation is often more complex: both nodes and links come with capacity constraints (e.g., in terms of buffers, CPU, and bandwidth) and may be attributed with costs (e.g., monetary or in terms of performance). In order to reason about performance, cost, and fairness aspects, it is therefore important to take these dimensions into account.

The challenge of extending NetKAT to weighted scenarios lies in the fact that in a weighted network, traffic flows can no longer be considered independently, but they may *interfere*:

their packets compete for the shared resource. Moreover, packets of a given flow may not necessarily be propagated along a unique path, but may be split and distributed among multiple paths (in the so-called *multi-path routing* or *splittable flow* variant). Accordingly, a weighted extension of NetKAT must be able to deal with "inter-packet states".

We in this paper will think of the network as a weighted (directed) graph $G = (V, E, w)$. Here, V denotes the set of switches (or equivalently routers, and henceforth often simply called nodes), E is the set of links (connected to the switches by *ports*), and w is a weight function. The weight function w applies to both nodes V as well as links E. Moreover, a node and a link may be characterized by *a vector of weights* and also combine *multiple resources*: for example, a list of capacities (e.g., CPU and memory on nodes, or bandwidth on links) and a list of costs (e.g., performance, energy, or monetary costs).

In order to specify the quantitative aspects, we propose in this paper a weighted extension of NetKAT: *WNetKAT*. In addition to NetKAT:

- *WNetKAT* includes a set of *quantitative packet-variables* to specify the quantitative information carried in the packet, in addition to the regular (non-quantitative) packet-variables of NetKAT (called *fields* in NetKAT): e.g., regular variables are used to describe locations, such as switch and port, or priorities, while quantitative variables are used to specify latency or energy. The set of all packet-variables is denoted by $\mathcal{V}_p$.
- *WNetKAT* also includes a set of *switch-variables*, denoted by $\mathcal{V}_s$, to specify the configurations at the (stateful) switch. Switch variables can either be quantitative (e.g., counters, meters, meta-rules [4, 33]) or non-quantitative (e.g., location related), as it is the case of the packet-variables.

▶ Remark. The set of quantitative (packet- and switch-) variables is denoted by $\mathcal{V}_q$ and these variables range over the natural numbers $\mathbb{N}$ (e.g., normalized rational numbers). The set of non-quantitative (packet- and switch-) variables is denoted $\mathcal{V}_n$ and the set of the possible values is denoted Ω. Note that $\mathcal{V}_q \cap \mathcal{V}_n = \emptyset$ and $\mathcal{V}_q \cup \mathcal{V}_n = \mathcal{V}_p \cup \mathcal{V}_s$.

In addition to introducing quantitative variables, we also need to extend the atomic actions and tests of NetKAT. Concretely, *WNetKAT* first supports non-quantitative assignments and non-quantitative tests on the non-quantitative switch-variables, similar to those on the packet-variables in NetKAT. Moreover, *WNetKAT* also allows for *quantitative assignments* and *quantitative tests*, defined as follows, where $x \in \mathcal{V}_q$, $\mathcal{V}' \subseteq \mathcal{V}_q$, $\delta \in \mathbb{N}$, $\bowtie \in \{>, <, \leq, \geq, =\}$:

- **Quantitative Assignment.** $x \leftarrow (\Sigma_{x' \in \mathcal{V}'} x' + \delta)$: Read the current values of the variables in $\mathcal{V}'$ and add them to δ, then assign this result to x.
- **Quantitative Test.** $x \bowtie (\Sigma_{x' \in \mathcal{V}'} x' + \delta)$: Read the current value of the variables in $\mathcal{V}'$ and add them to δ, then compare this result to the current value of x.

▶ Remark.

1. In the quantitative assignment and test, only addition is allowed. However, an extension to other arithmetic operations (e.g., linear combinations) is straightfoward. Moreover, calculating *minimum* or *maximum* may be useful in practice: e.g., the throughput of a flow often depends on the weakest link (of minimal bandwidth) along a path. Note that these operations can actually be implemented with quantitative assignments and tests, i.e., by comparing every variable to another and determining the smallest. E.g., for $x \in \mathcal{V}_q$ and $y, z \in \mathcal{V}_q$ or $\mathbb{N}$,

$$x \leftarrow \min\{y, z\} \overset{\text{def}}{=} y \leq z; x \leftarrow y \;\&\; y > z; x \leftarrow z\,.$$

2. In quantitative assignment and test, x might be in $\mathcal{V}'$.
3. We use $+$ to denote the arithmetic operation over numbers. Therefore, we will use "&" in *WNetKAT* to denote the "+" operator of Kleene Algebra, which is also used in [14].

Table 1 Semantics of *WNetKAT*.

$$[\![x \leftarrow \omega]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk[\omega/x] :: h\} & \text{if } x \in \mathcal{V}_p \\ \{\rho(v)[\omega/x],\ pk :: h\} & \text{if } x \in \mathcal{V}_s \text{ and } pk(sw) = v \end{cases} \quad (1)$$

$$[\![x = \omega]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk :: h\} & \text{if } x \in \mathcal{V}_p \text{ and } pk(x) = \omega \\ & \text{or if } x \in \mathcal{V}_s, pk(sw) = v \text{ and } \rho(v,x) = \omega \\ \emptyset & \text{otherwise} \end{cases} \quad (2)$$

$$[\![y \leftarrow (\Sigma_{y' \in \mathcal{V}'} y' + r)]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk[r'/x] :: h\} & \text{if } x \in \mathcal{V}_p \\ \{\rho(v)[r'/x],\ pk :: h\} & \text{if } x \in \mathcal{V}_s \text{ and } pk(sw) = v \end{cases} \quad (3)$$
$$\text{where } r' = \Sigma_{y_p \in \mathcal{V}' \cap \mathcal{V}_p} pk(y_p) + \Sigma_{y_s \in \mathcal{V}' \cap \mathcal{V}_q} \rho(v, y_s) + r$$

$$[\![y = (\Sigma_{y' \in \mathcal{V}'} y' + r)]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk :: h\} & \text{if } x \in \mathcal{V}_p \text{ and } pk(x) = r' \\ & \text{or } x \in \mathcal{V}_s, pk(sw) = v \text{ and } \rho(v,x) = r' \\ \emptyset & \text{otherwise} \end{cases} \quad (4)$$
$$\text{where } r' = \Sigma_{y_p \in \mathcal{V}' \cap \mathcal{V}_p} pk(y_p) + \Sigma_{y_s \in \mathcal{V}' \cap \mathcal{V}_q} \rho(v, y_s) + r$$

Given the set of switches V, a *switch-variable valuation* is a partial function $\rho : V \times \mathcal{V}_s \hookrightarrow \mathbb{N} \cup \Omega$. It associates, for each switch and each switch-variable, a integer or a value from Ω. We emphasize that ρ is a partial function, as some variables may not be defined at some switches.

A *WNetKAT*expression denotes a function $[\![\ ]\!] : \rho \times H \to 2^H$, where H is the set of packet histories. The semantics of *WNetKAT* is defined in Table 1, where $x \in \mathcal{V}_n, y \in \mathcal{V}_q$, $\delta \in \mathbb{N}$ and $\omega \in \Omega$.

▶ Remark.

- Equations (1) and (3) update the corresponding header field if x is a packet-variable, or they update the corresponding switch information of the current switch if x is a switch-variable. Equation (1) updates the non-quantitative variables and Equation (3) the quantitative ones.
- Equations (2) and (4) test the non-quantitative and quantitative variables respectively, using the current packet- and switch-variables.

▶ **Example 2.** Consider again the network in Figure 1. The topology of the network can be characterized with the following *WNetKAT* formula t, where sw specifies the current location (switch) of the packet, co specifies the cost, and ca specifies the capacity along the links.

$$\begin{aligned} t ::= \quad & sw = s; (sw \leftarrow F_1; co \leftarrow co + 1; ca \leftarrow \min\{ca, 8\} \\ & \qquad \& \ sw \leftarrow v; co \leftarrow co + 5; ca \leftarrow \min\{ca, 2\}) \\ & \& \ sw = F_1; \\ & \quad (sw \leftarrow F_2^{(1)}; co \leftarrow co + 3; ca \leftarrow \min\{ca, 1\} \\ & \quad \& \ sw \leftarrow F_2^{(2)}; co \leftarrow co + 2; ca \leftarrow \min\{ca, 10\}) \\ & \& \ sw = v; (sw \leftarrow F_2^{(1)}; co \leftarrow co + 3; ca \leftarrow \min\{ca, 3\} \\ & \qquad \& \ sw \leftarrow F_2^{(2)}; co \leftarrow co + 2; ca \leftarrow \min\{ca, 1\}) \\ & \& \ sw = F_2^{(1)}; sw \leftarrow t; co \leftarrow co + 6; ca \leftarrow \min\{ca, 1\} \\ & \& \ sw = F_2^{(2)}; sw \leftarrow t; co \leftarrow co + 1; ca \leftarrow \min\{ca, 4\} \end{aligned}$$

The variable co accumulates the costs along the path, and the variable ca records the smallest capacity along the path. Notice that ca is just a packet-variable used to record the capacity of the path; it does not represent the capacity used by this packet (the latter is assumed to be negligible).

Assume that function F_1 is flow conserving (e.g., a NAT), while F_2 increases the flow rate by an additive constant $\gamma \in \mathbb{N}$ (e.g., a security related function, adding a watermark or

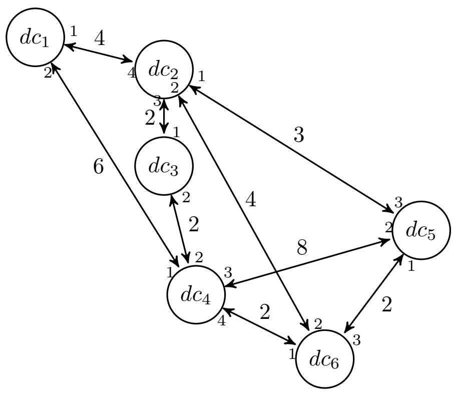

Figure 2 Example topology: excerpt of Google B4 [16] (U.S. data centers only). Nodes here represent data centers (resp. OpenFlow switches located at the end of the corresponding long-haul fibers). Links are annotated with weights, and nodes are interconnected via ports (*small numbers*).

an IPSec header). The policy of F_2 can be specified as:

$$p_{F_2} ::= \quad (sw = F_2^{(1)} \ \& \ sw = F_2^{(2)}); ca \leftarrow ca + \gamma$$

▶ Remark. Note that this simple example required only (non-quantitative and quantitative) packet-variables. However, as we will see in Section 4, to model more complex aspects of networking, such as splittable flows, additonal concepts of *WNetKAT* will be needed.

4 Applications

The weighted extensions introduced by *WNetKAT* come with a number of interesting applications. In this section, we show that the notions of reachability frequently discussed in prior work, find natural extensions in the world of weighted networks, and discuss applications in the context of service chains, fairness, and quality-of-service. In our technical report [25], additional details are provided for some of these use cases.

4.1 Cost Reachability

Especially data center networks but also wide-area networks, and to some extent enterprise networks, feature a certain *path diversity* [41]: there exist multiple routes between two endpoints (e.g., hosts). This path diversity is not only a prerequisite for fault-tolerance, but also introduces traffic engineering flexibilities. In particular, different paths or routes depend on different links, whose cost can vary. For example, links may be attributed with monetary costs: a peering link may be free of charge, while an up- or down-link is not. Links cost can also be performance related, and may for example vary in terms of latency, for example due to the use of different technologies [37], or simply because of different physical distances. The monetary and performance costs are often related: for example, in the context of stock markets, lower latency links come at a higher price [35]. It is therefore natural to ask questions such as: *"Can A reach B at cost at most c?"*. We will refer to this type of questions as *cost reachability questions*.

▶ **Example 3.** Consider the network in Figure 2. The topology roughly describes the North American data centers interconnected by Google B4, according to [16].

In order to reason about network latencies, we not only need information about the switch at which the packet is currently located (as in our earlier examples), but also the *port of the switch* needs to be specified. We introduce the packet-variable *pt*. We can then specify this network topology in *WNetKAT*. The link from dc_1 to dc_2 (latency 4 units) represented by the port 1 at dc_1 and the port 4 at dc_2 is specified as follows, where we use packet-variable *sw* to denote the current switch, *pt* to specify the current port, and l to specify the latency of the path the packet traverses,

$$sw = dc_1; pt = 1; sw \leftarrow dc_2; pt \leftarrow 4; l \leftarrow l + 4\,.$$

Analogously, the entire network topology can be modeled with *WNetKAT*, henceforth denoted by t. The policy of the network determines the functionality of each switch (the OpenFlow rules), e.g., in dc_2, packets from dc_1 to dc_5 arriving at port 4 are always sent out through port 1 or port 3. This can be specified as:

$$src = dc_1; dst = dc_5; sw = dc_2; pt = 4; (pt \leftarrow 1 \&\ pt \leftarrow 3)\,.$$

Analogously, the entire network policy can be modeled with *WNetKAT*, henceforth denoted by p.

To answer the cost reachability question, one can check whether the following *WNetKAT* expression is equal to *drop*.

$$scr \leftarrow A; dst \leftarrow B; l \leftarrow 0; sw \leftarrow X; pt(pt)^*; sw = B; l \leq c\,.$$

If it is equal to *drop*, then B cannot be reached from A at latency at most c; otherwise, it can.

▶ Remark. For ease of presentation, in the above example, we considered only one weight. However, *WNetKAT* readily supports multiple weights: we can simply use multiple variables accordingly. Moreover, while the computational problem complexity can increase with the number of considered weights [23], the multi-constrained path selection does not affect the general asymptotic complexity of *WNetKAT*.

4.2 Capacitated Reachability

Especially in the wide-area network, but also in data centers, link capacities are a scarce resource: indeed, wide-area traffic is one of the fastest growing traffic aggregates [16]. However, also the routers themselves come with capacity constraints, both in terms of memory (size of TCAM) as well as CPU: for example, the CPU utilization has been shown to depend on the packet rate [31]. Accordingly, a natural question to ask is: *Can A communicate at rate at least r to B?* We will refer to this type of questions as *capacitated reachability questions.*

There are two problem variants:

- *Unsplittable flows:* The capacity needs to be computed along a single path (e.g., an MPLS tunnel).
- *Splittable flows:* The capacity needs to be computed along multiple paths (e.g., MPTCP, ECMP). We will assume links of higher capacity are chosen first.

For both variants, to find out the capacity of paths between two nodes, a *single* test packet will be sent to explore the network and record the bandwidth/capacity with a packet-variable in the packet. We assume that the bandwidth consumed by this packet is negligibile. Also, only once the packet has traversed and determined the bandwidth, e.g., the actual (large) flows are allocated accordingly (by the SDN controller).

▶ **Example 4.** Consider the network in Figure 2 again, but assume that the labels are the capacities rather than latency.

Unsplittable flow scenario: The switch policies are exactly the same as in Example 3, while the topology will be specified similarly using packet-variable c to record the capacity of the link. E.g., the link between dc_1 and dc_2 can be specified as:

$$sw = dc_1; pt = 1; sw \leftarrow dc_2; pt \leftarrow 4; c \leftarrow \min\{c, 4\}\,.$$

The unsplittable capacitated reachability question can be answered by checking whether the following expression is equal to *drop*,

$$scr \leftarrow A; dst \leftarrow B; c \leftarrow r; sw \leftarrow A; pt(pt)^*; sw = B; c \geq r\,.$$

If the above formula does not equal *drop*, then A can communicate at rate at least r to B.

Another (possibly) more efficient approach is not to update c while the bandwidth is smaller than r (meaning that a flow of size r cannot go through this link). In this case, one can specify the topology as follows, where c is not used to record the capacity along the path anymore, but rather to test whether this link is wide enough:

$$sw = dc_1; pt = 1; sw \leftarrow dc_2; pt \leftarrow 4; c \leq 4\,.$$

The above *WNetKAT* expression only tests whether c is less than or equal to 4. It makes sure that the value of c (which is r) does not exceed the capacity of the following link. If it exceeds the capacity of the link, then a flow of rate r cannot use this link. Therefore, the test packet is dropped already. The capacitated reachability question can then be answered by checking whether the following expression is equal to *drop*:

$$scr \leftarrow A; dst \leftarrow B; c \leftarrow r; sw \leftarrow A; pt(pt)^*; sw = B\,.$$

If the above formula does not equal *drop*, then A can communicate at rate at least r to B.

Splittable flow scenario: For the splittable scenario, the situation is far more complicated. For example, in dc_2, packets arriving at port 4 are sent out through port 1 or port 2, and port 2 prioritizes port 1. That is, if the incoming traffic has rate 4, then a share of 3 units will be sent out through port 2, and a 1 share through port 1.

Note that also here, still only one *single* test packet will be sent to collect the capacity information. This information will be stored in the packet-variable c as well. However, when the test packet arrives at a switch where a flow can be split, copies of the packet are sent (after updating the c according to the bandwidth of each path) to all possible paths, to record the capacity along all other paths. This exploits the fact that *WNetKAT* (NetKAT) treats the & operator as *conjunction* in the sense that both operations are performed, rather than *disjunction*, where one of the two operations would be chosen non-deterministically (according to the usual Kleene interpretation). Again, we emphasize that we will refer to c stored in one *single* test packet, and not the actual real data flow. Now the topology will update c as in the unsplittable case. However, the policy needs to not only decide which ports the packets go to, but also update c according to the split policy. E.g.„ at dc_2, the data flow from dc_1 to dc_5 at rate 4 is sent out through port 1 at rate 3, and the port 3 at 1. And if the rate is smaller than or equal to 3, e.g., 2, then the whole flow of rate 2 will be sent out through port 1. The following *WNetKAT* formula specifies this behavior:

$$\begin{aligned}
&src = dc_1; dst = dc_5; sw = dc_2; pt = 4; c \leq 5\\
&\qquad (pt \leftarrow 1; c \leftarrow \min\{3, c\}\\
&\qquad \&\ pt \leftarrow 3; c \leftarrow \max\{0, c - 3\})
\end{aligned}$$

The test $c \leq 5$ ensures that the flow does not exceed the capacity of both paths. Notice that even when the size of the flow is small enough for one path, a copy of the test packet with $c = 0$ will still be sent to the other. This ensures that sufficient information is available at the switch where flows merge. That is, the switch collects the weights the packets carry (c in our example). The switch will only push packets to the right out-ports after all expected packets have arrived. This will happen before the switch sends the packet to the right out-ports. For example, at dc_4, the flow from dc_1 to dc_5 might arrive in from ports 1 and 2 and will be sent out through port 3. In order to record the capacity of both links, switch-variables C and X are introduced, for each possible merge. For example, the following table provides the merging rules for the switch at dc_4, where X is the counter for the merge, and C stores the current capacity of the arriving test packets. Initially, X is set to the number of in-ports for the merge, and C is set to 0.

src	dst	in	out	C	X
dc_1	dc_5	$1, 2$	3	0	2
dc_5	dc_2	$3, 4$	$1, 2$	0	2

The first line of the rules in the table can be specified in *WNetKAT* as follows:

$$\begin{aligned} & sw = dc_4; src = dc_1; dst = dc_5; (pt = 1 \ \& \ pt = 2); \\ & \qquad C \leftarrow C + c; X \leftarrow X - 1; \\ & \quad (X \neq 0; drop \ \& \ \ X = 0; c \leftarrow C; pt \leftarrow 3) \end{aligned}$$

When a packet from dc_1 to dc_5 arrives at port 1 or 2 of dc_4, first the switch collects the value of c and adds it to the switch-variable C, then decrements X to record that one packet arrived. Afterwards, we test whether all expected packets arrived ($X = 0$). If not, the current one is dropped; if yes, we send the current packet out to port 3. The reason that we can drop all packets except for the last, is that all those packets carry exactly the same values. Therefore, we eventually only need to include the merged capacity (C) in the last packet, and propagate it.

Combining the split and merge cases, the policy of the switch can be defined. For example, the second line of the merging rule table can be specified as follows, by first merging from port 3 and 4, and then splitting to port 1 and 2:

$$\begin{aligned} & sw = dc_4; src = dc_5; dst = dc_2; (pt = 3 \ \& \ pt = 4); \\ & \quad C \leftarrow C + c; X \leftarrow X - 1; \\ & \quad (X \neq 0; drop \ \& \ X = 0; c \leftarrow C; c \leq 8 \\ & \qquad\qquad\qquad (pt \leftarrow 1; c \leftarrow \min\{6, c\} \\ & \qquad\qquad\qquad \& \ pt \leftarrow 2; c \leftarrow \max\{0, c - 6\})) \end{aligned}$$

Then the splittable capacited reachability question can be answered by checking whether the following expression evaluates to *drop*:

$$\begin{aligned} & scr \leftarrow A; dst \leftarrow B; c \leftarrow r; sw \leftarrow A; pt(pt)^*; \\ & \qquad\quad sw = B; X = 0; c \geq r \end{aligned}$$

If the above formula does not equal *drop*, then A can communicate at rate at least r to B.

4.3 Service Chaining

The virtualization and programmability trend is not limited to the network, but is currently also discussed intensively for network functions in the context of the Network Function

Virtualization (NFV) paradigm. SDN and NFV nicely complement each other, enabling innovative new network services such as *service chains* [17]: network functions which are traversed in a particular order (e.g., first firewall, then cache, then wide-area network optimizer). Our language allows to reason about questions such as *Are sequences of network functions traversed in a particular order, without violating node and link capacities?* *WNetKAT* can easily be used to describe weighted aspects also in the context of service chains. In particular, network functions may both *increase* (e.g., due to addition of an encapsulation header, or a watermark) or *decrease* (e.g., a WAN optimizer, or a cache) the traffic rate, both *additively* (e.g., adding a header) or *multiplicatively* (e.g., WAN optimizer).

▶ **Example 5.** Let us go back to Figure 1, and consider a service chain of the form (s, F_1, F_2, t): traffic from s to t should first traverse a function F_1 and then a function F_2, before reaching t. For example, F_1 may be a firewall or proxy and F_2 is a WAN optimizer. The virtualized functions F_1 and F_2 may be allocated redundantly and may change the traffic volume. Using *WNetKAT*, we can ask questions such as: *What is the maximal rate at which s can transmit traffic into the service chain?* or *Can we realize a service chain of cost (e.g., latency) at most x?*. Let us consider the following example: The question *"Can s reach t at cost/latency at most ℓ and/or at rate/bandwidth at least r, via the service chain functions F_1 and F_2?"*, can be formulated by combining the reachability problems above and the waypointing technique in [1]. For example, in case of cost reachability, we can ask if the following *WNetKAT* formula equals *drop*.

$$\begin{aligned}
&src \leftarrow s; dst \leftarrow t; co \leftarrow 0; sw \leftarrow s; pt(pt)^*;\\
&\quad sw = F_1; p_{F_1}; tpt(pt)^*; sw = F_2; p_{F_2};\\
&\qquad tpt(pt)^*; sw = t; co \leq \ell; ca \geq r
\end{aligned}$$

Note that in this example, we considered an unsplittable scenario. For the splittable scenario, we can extend the splittable capacitated reachability use case above analogously.

5 (Un)Decidability

In this section we shed light on the fundamental decidability of weighted SDN programming languages like WNetKAT. Given today's trend toward more quantitative networking, we believe that this is an important yet hardly explored dimension. In particular, we will establish an equivalence between WNetKAT and weighted automata.

In the following, we will restrict ourselves to settings where quantitative variables of the same type behave similarly in the entire network: For example, the cost variables (e.g., quantifying latencies) in the network are always added up along a given path, while capacity variables require minimum operations along different paths. This is a reasonable for real-world networks.

The definition of the weighted automata used here is slightly different from those usually studied, e.g., [6, 9]. However, it is easy to see that they are equivalent.

We first introduce some preliminaries. A *semiring* is a structure $(K, \oplus, \otimes, 0, 1)$, where $(K, \oplus, 0)$ is a commutative monoid, $(K, \otimes, 1)$ is a monoid, multiplication distributes over addition $k \otimes (k' \oplus k'') = k \otimes k' \oplus k \otimes k''$, and $0 \otimes k = k \otimes 0 = 0$ for each $k \in K$. For example, $(\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ and $(\mathbb{N} \cup \{\infty\}, \max, +, \infty, 0)$ are semirings, named the *tropical semiring*. $(\mathbb{N} \cup \{\infty\}, \max, \min, 0, \infty)$ is also a semiring. A *bimonoid* is a structure $(K, \oplus, \otimes, 0, 1)$, where $(K, \oplus, 0)$ and $(K, \otimes, 1)$ are monoids. K is called a *strong bimonoid* if $\oplus$ is commutative and $0 \otimes k = k \otimes 0 = 0$ for each $k \in K$. For example, $(\mathbb{N} \cup \{\infty\}, +, \min, 0, \infty)$ is a (strong) bimonoid, named the *tropical bimonoid*.

Now fix a semiring/bimonoid K and an alphabet Σ. A *weighted finite automaton* (WFA) over K and Σ is a quadruple $A = (S, s, F, \mu)$ where S is a finite set of states, s is the starting state, F is set of the final states, $\mu : \Sigma \to K^{S\times S}$ is the transition weight function and λ is the weight of entering the automaton. For $\mu(a)(s, s') = k$, we write $s \xrightarrow{a}_k s'$.

Let At be the set of complete non-quantitative tests and P be the set of complete non-quantitative assignments. Let Ω be the set of complete quantitative tests and Δ be the set of complete quantitative assignments.

A weighted NetKAT automata is a finite state weighted automaton $A = (S, s, F, \lambda, \mu)$ over a structure K and alphabet Σ. The inputs to the automaton are so called reduced strings introduced in [1, 15], which belong to the set $U = \mathsf{At} \cdot \Omega \cdot \mathsf{P} \cdot \Delta \cdot (\mathsf{dup} \cdot \mathsf{P} \cdot \Delta)^*$, i.e., the strings belonging to U are of the form:

$$\alpha\omega p_0\delta_0 \ \mathsf{dup}\ \mathsf{p}_1\delta_1 \ \mathsf{dup}\ \cdots\ \mathsf{dup}\ \mathsf{p}_\mathsf{n}\delta_\mathsf{n}$$

for some $n \geq 0$. Intuitively, μ attempts to consume $\alpha\omega p_0\delta_0$ dup from the front of the input string and move to a new state with a weight and the new state has the residual input string $\alpha_0\omega_0\ p_1\delta_1$ dup $\cdots$ dup $\mathsf{p}_\mathsf{n}\delta_\mathsf{n}$.

The following construction shows the equivalence between WNetKAT and weighted automata.

From WFA to WNetKAT. Let $A = (S, s, F, \lambda, \mu)$ be a weighted NetKAT automata over K and Σ. An accepting path in A $s \xrightarrow{r_1}_{\alpha_1\beta_1} s_1 \xrightarrow{r_2}_{\alpha_2\beta_2} s_2 \cdots \xrightarrow{r_n}_{\alpha_n\beta_n} s_n$ can be write as the following WNetKAT expression:

$$\alpha_1\omega_1 p_1\delta_1\ \mathsf{dup}\ p_2\delta_2\ \mathsf{dup}\ \cdots\ \mathsf{dup}\ p_n\delta_n\ ,$$

where

1. $\omega_1 = \lambda$, $\delta_1 = \omega_1 \oplus r$ and $\delta_i = \delta_{i-1} \oplus r_i$ for $i = 2, ..., n$;
2. $p_i = p_{\beta_i}$ for $i = 1, ..., n$.

From WNetKAT to WFA. Let e be a weighted automata expression, then following [1, 15], we can define a set of reduced strings R which are semantically equivalent to e. We define a weighted NetKAT automata $A = (S, s, F, \lambda, \mu)$ over a structure K and alphabet Σ, where

- $s = R$ and $\Sigma = \mathsf{At} \times \mathsf{At}$.
- $\mu : \Sigma \to K^{S\times S}$ is defined as: $\mu(\alpha, \beta)(u_1, u_2) = r$ iff $u_2 = \{\beta\omega' x \mid \alpha\omega p\delta\ \mathsf{dup}\ x \in u_1\}$, where $\beta = \alpha_p, \omega' = \delta_\omega$ and $\omega \otimes r = \omega'$. For short write $u_1 \xrightarrow{r}_{\alpha\beta} u_2$.
- $S = \{s\} \cup \{u \subseteq 2^U \mid \exists\ \mu\text{-path } s \to \cdots \to u\}$.
- $F = \{u \mid \alpha\omega p\delta \in u \in S\}$.
- $\lambda = \{\omega \mid \alpha\omega x \in s\}$.

We have the following theorem.

▶ **Theorem 6.**

1. *For every finite weighted WNetKAT automaton A, there exists a WNetKAT expression e such that the set of reduced strings accepted by A is the set of reduced strings of e.*
2. *For every WNetKAT expression e, there is a weighted WNetKAT automaton A accepting the set of the reduced strings of e.*

Let us just give some examples:

1. For the cost reachability use case, there exists a weighted WNetKAT automaton over the tropical semiring $(\mathbb{N} \cup \{\infty\}, +, \min, \infty, 0)$ that accepts the set of reduced strings of the WNetKAT expression in Section 4.1.

2. For the capacitated reachability: (i) There exists a weighted WNetKAT automaton over the semiring $(\mathbb{N} \cup \{\infty\}, \max, \min, 0, \infty)$ that accepts the set of the reduced strings of the WNetKAT expression for the splitable case in Section 4.2. (ii) There exists a weighted WNetKAT automaton over the tropical bimonoid $(\mathbb{N} \cup \{\infty\}, \min, +, 0, \infty)$ that accepts the set of the reduced strings of the WNetKAT expression for the unsplitable case in Section 4.2.

From this relationship, we have the following theorem about the (un)decidability of WNetKAT expression equivalence.

▶ **Theorem 7.** *Deciding equivalence of two WNetKAT expressions is equal to deciding the equivalence of the two corresponding weighted WNetKAT automata.*

For all the semiring and bimonoid we encountered in this paper, the WFA equivalence is undecidable. Therefore, the equivalence is also undeciable.

This negative result highlights the inherent challenges involved in complex network languages which are powerful enough to deal with weighted aspects.

However, we also observe that in many practical scenarios, the above undecidability result is too general and does not apply. For example, most of the use cases presented in in Section 4 can actually be reduced to test *emptiness*: we often want to test whether a given WNetKAT expression e equals 0, i.e., whether the corresponding weighted NetKAT automaton is empty. Indeed, there seems to exist an intriguing relationship between emptiness and reachability.

▶ **Theorem 8.** *Deciding whether a WNetKAT expression is equal to* 0 *is equal to deciding the emptiness of the corresponding weighted automaton.*

Interestingly, as shown in [7, 8, 21, 22], the emptiness problem is decidable for several semirings/bimonoids, e.g., the tropical semiring and the tropical bimonoid used in this paper. This leads to the decidability of the WNetKAT equivalence over these structures.

Another interesting domain with many decidability results are unambiguous regular grammars and unambiguous finite automata [40]. Accordingly, in our future work, we aim to extend these concepts to the weighted world and explore the unambiguous subsets of WNetKAT which might enable decidability for equivalence.

6 Conclusion

While OpenFlow today does not per se accommodate *stateful* packet operations or support arithmetic computations, we currently witness a trend toward computationally more advanced and stateful packet-processing functionality, see e.g., P4 or OpenState. Moreover, in order to implement arithmetic operations (see e.g., Equations (3) and (4)), we can simply use lookup tables realized as OpenFlow rules, see the technique in [32]. For a simple yet inefficient solution to compile *WNetKAT* switch variables is to use round robin groups [32].

In our future research, we aim to chart a more comprehensive landscape of the decidability and decision complexity of *WNetKAT* in different settings, and related to this, investigate the axiomatization in more depth. On the practical side, we are exploring possibilities for compiling *WNetKAT* to (extended) OpenFlow protocols such as P4 [4].

Finally, we refer the reader to our technical report [25] for additional details and use cases.

Acknowledgements. We would like to thank Alexandra Silva, Nate Foster, Dexter Kozen, Manfred Droste and Fredrik Dahlqvist for many inputs and discussions on WNetKAT.

References

1 Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.*, 49(1), January 2014. doi:10.1145/2578855.2535862.

2 Ryan Beckett, Michael Greenberg, and David Walker. Temporal NetKAT. In *Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 386–401, 2016.

3 Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.*, 44(2), April 2014.

4 Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM CCR*, 44(3):87–95, 2014. doi:10.1145/2656877.2656890.

5 Kenneth L. Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz. Directions in active networks. *Communications Magazine, IEEE*, 36(10):72–78, 1998.

6 Manfred Droste and Paul Gastin. Weighted automata and weighted logics. In *Proc. ICALP*, 2005.

7 Manfred Droste and Doreen Götze. The support of nested weighted automata. In *Proc. Workshop on Non-Classical Models for Automata and Applications – (NCMA)*, 2013.

8 Manfred Droste and Doreen Heusel. The supports of weighted unranked tree automata. *Fundam. Inform.*, 2015.

9 Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata*. Springer Science & Business Media, 2009.

10 Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 217–232, 2016.

11 Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN. *Queue*, 11(12):20:20–20:40, December 2013.

12 Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proc. ACM SIGCOMM*, pages 327–338, 2013.

13 Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proc. 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 279–291, 2011.

14 Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In *Proc. ESOP*, 2016.

15 Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *ACM SIGPLAN Notices*, 2015.

16 Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Stephen Stuart Jonathan Zolla, Urs Hölzle, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. *SIGCOMM Comput. Commun. Rev.*, 43(4), 2013. doi:10.1145/2486001.2486019.

17 Wolfgang John, Konstantinos Pentikousis, George Agapiou, Eduardo Jacob, Mario Kind, Antonio Manzalini, Fulvio Risso, Dimitri Staessens, Rebecca Steinert, and Catalin Meirosu. Research directions in network service chaining. In *Proc. IEEE SDN for Future Networks and Services*, 2013. doi:10.1109/SDN4FNS.2013.6702549.

18 Garvit Juniwal, Nikolaj Bjorner, Ratul Mahajan, Sanjit Seshia, and George Varghese. Quantitative network analysis. *Technical Report*, 2016.

19 Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proc. USENIX NSDI*, 2012.

20 Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proc. USENIX NSDI*, 2013.

21 Daniel Kirsten. The support of a recognizable series over a zero-sum free, commutative semiring is recognizable. In *Developments in Language Theory, 13th International Conference, DLT 2009, Stuttgart, Germany, June 30 – July 3, 2009. Proceedings*, pages 326–333, 2009.

22 Daniel Kirsten and Karin Quaas. Recognizability of the support of recognizable series over the semiring of the integers is undecidable. *Inf. Process. Lett.*, 111(10):500–502, 2011.

23 Turgay Korkmaz and Marwan Krunz. Multi-constrained optimal path selection. In *Proc. IEEE INFOCOM 2001*, volume 2, pages 834–843, 2001.

24 Dexter Kozen. *Kleene algebra with tests and commutativity conditions.* Springer, 1996.

25 Kim G. Larsen, Stefan Schmid, and Bingtian Xue. WNetKAT: A Weighted SDN Programming and Verification Language. In *ArXiv technical report 1608.08483*, 2016.

26 Tamas Lukovszki and Stefan Schmid. Online admission control and embedding of service chains. In *Proc. SIROCCO*, 2015.

27 Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proc. ACM SIGCOMM*, 2011. doi:10.1145/2018436.2018470.

28 Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *ACM SIGPLAN Notices*, 2012.

29 Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proc. USENIX NSDI*, pages 1–14, 2013.

30 Oded Padon, Neil Immerman, Aleksandr Karbyshev, Ori Lahav, Mooly Sagiv, and Sharon Shoham. Decentralizing SDN policies. In *ACM SIGPLAN Notices*, 2015.

31 M. Paredes-Farrera, M. Fleury, and M. Ghanbari. Router response to traffic at a bottleneck link. In *Proc. TRIDENTCOM*, 2006.

32 Liron Schiff, Michael Borokhovich, and Stefan Schmid. Reclaiming the brain: Useful openflow functions in the data plane. In *Proc. ACM HotNets*, 2014.

33 Liron Schiff, Petr Kuznetsov, and Stefan Schmid. In-Band Synchronization for Distributed SDN Control Planes. *Proc. ACM SIGCOMM CCR*, 2016.

34 Cole Schlesinger, Hitesh Ballani, Thomas Karagiannis, and Dimitrios Vytiniotis. Quality of service abstractions for software-defined networks. *Technical Report*, 2016.

35 David Schneider. The microsecond market. In *Proc. IEEE Spectrum*, 2012.

36 Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM 2012*, 2012. doi:10.1145/2342356.2342359.

37 Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. The internet at the speed of light. In *Proc. ACM HotNets-XIII*, 2014.

38 Jonathan M. Smith and Scott M. Nettles. Active networking: one view of the past, present, and future. *Proc. IEEE Transactions on Systems, Man, and Cybernetics: Applications and Reviews*, 34(1):4–18, 2004.

39 Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proc. ACM CoNEXT*, pages 213–226, 2014.

40 Richard E. Stearns and Harry B. Hunt. On the equivalence and containment problems for unambiguous regular expressions, grammars, and automata. In *Proc. 22nd Annual Symposium on Foundations of Computer Science (SFCS)*, 1981.

41 Renata Teixeira, Keith Marzullo, Stefan Savage, and Geoffrey M. Voelker. Characterizing and measuring path diversity of internet topologies. In *ACM SIGMETRICS PER*, 2003. doi:10.1145/885651.781069.

42 Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. Some complexity results for stateful network verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2016.

43 Andreas Voellmy, Ashish Agarwal, and Paul Hudak. Nettle: Functional reactive programming for openflow networks. Technical report, Yale University, 2010.

44 Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: simplifying SDN programming using algorithmic policies. In *SIGCOMM CCR*, 2013. doi:10.1145/2534169.2486030.

45 Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally verifiable networking. *Proc. ACM HotNets*, 2009.

46 Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. Enforcing customizable consistency properties in software-defined networks. In *Proc. USENIX NSDI*, 2015.