NetKAT — A Formal System for the Verification of Networks

Dexter Kozen

Cornell University, Ithaca, NY 14853-7501, USA
kozen@cs.cornell.edu
http://www.cs.cornell.edu/~kozen

Abstract. This paper presents a survey of recent work in the development of NetKAT, a formal system for reasoning about packet switching networks, and its role in the emerging area of software-defined networking.

Keywords: Kleene algebra, Kleene algebra with tests, NetKAT, software defined networking, packet switching, OpenFlow, Frenetic

1 Introduction

NetKAT is a relatively new language and logic for reasoning about packet switching networks. The system was introduced quite recently by Anderson et al. [1] and further developed by Foster et al. [10]. The present paper provides an accessible self-contained introduction to the NetKAT language, some examples of things one can do with it, and a flavor of ongoing work. All the results described here have appeared previously [1, 10].

1.1 Software-Defined Networking

Traditional network architecture is fairly low-level, consisting of routers and switches that do little besides maintaining routing tables and forwarding packets. The components of the network are typically configured locally, making it difficult to implement end-to-end routing policies and optimizations that require a global perspective. This state of affairs is ill-suited to modern data centers and cloud-based applications that require a higher degree of coordination among network components to function effectively.

Software-defined networking (SDN) is a relatively new paradigm for network management. The main idea behind SDN is to permit centralized control of the network in the form of a controller that communicates with the individual network components. As the Open Networking Foundations's 2012 white paper "Software-Defined Networking: The New Norm for Networks" [11] describes it,

> In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications. As a

result, enterprises and carriers gain unprecedented programmability, automation, and network control, enabling them to build highly scalable, flexible networks that readily adapt to changing business needs.

One can think of a centralized controller or set of controllers that have global knowledge of the topology of the network over which they exercise control and can interact with individual network components via a standardized communication interface. The controller can receive traffic flow information and operational status from the components and can reconfigure them on the fly if necessary to balance load, reroute traffic to circumvent failures, or implement security policies.

1.2 NetKAT

NetKAT is a new domain-specific language and logic for specifying and verifying network packet-processing functions that fits well with the SDN paradigm. It is part of the Frenetic suite of network management tools [9, 12, 28, 29]. NetKAT is based on Kleene algebra with tests (KAT), a generic algebraic system for reasoning about partial correctness that has been studied since the 1990's [23]. KAT, in turn, is based on Kleene algebra (KA), the algebra of regular expressions [19]. NetKAT is essentially KAT with primitives for modifying and testing packet headers and encoding network topologies along with axioms for reasoning about those constructs.

One might at first be skeptical about the expressive power of regular expressions in this context, but in fact regular expressions are sufficient to encode network topology and express many common reachability and security queries, which can now be verified automatically. In §3 we give some examples of the types of queries one can express with NetKAT. This expressive power, coupled with NetKAT's formal mathematical semantics, complete deductive system, and decision procedure, make NetKAT a viable tool for SDN programming and verification.

2 NetKAT Basics

In this section we describe the syntax and semantics of NetKAT. This requires us to say a few words about Kleene algebra (KA) [19] and Kleene algebra with tests (KAT) [23] on which NetKAT is based.

2.1 Kleene Algebra (KA)

Kleene algebra is the algebra of regular expressions. Regular expressions are normally interpreted as regular sets of strings, but there are many other useful interpretations: binary relation models used in programming language semantics, the $(\min, +)$ algebra used in shortest path algorithms, models consisting of convex sets used in computational geometry. Perhaps surprisingly, a formal model of packet-switching networks can also be added to this list.

Abstractly, a *Kleene algebra* 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:

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

where we define $p \leq q$ iff $p + q = q$. The axioms above not involving * are succinctly stated by saying that the structure is an idempotent semiring under $+, \cdot, 0$, and 1, the term *idempotent* referring to the axiom $p + p = p$. Due to this axiom, the ordering relation $\leq$ is a partial order. The axioms for * together say that p^*q is the $\leq$-least solution of $q + px \leq x$ and qp^* is the $\leq$-least solution of $q + xp \leq x$.

One of the nice things about KA is that all properties are expressed as equations and equational implications (Horn formulas), and reasoning is purely equational. No specialized syntax or rules are needed, only the axioms and rules of classical equational logic. This is also true of KAT and NetKAT.

2.2 Kleene Algebra with Tests (KAT)

To get KAT from KA, we add Boolean tests. Formally, a KAT is a two-sorted structure $(K, B, +, \cdot, ^*, ^-, 0, 1)$, where $B \subseteq K$ and

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

The elements of B are called *tests*. Note that the semiring operations $+, \cdot, 0, 1$ are heavily overloaded, but this does not create any conflict. On tests, $+$ and $\cdot$ behave as Boolean disjunction and conjunction, respectively, and 0 and 1 stand for falsity and truth, respectively. The overline $^-$ is the Boolean negation operator, sometimes written as a prefix operator $\neg$.

The axioms of Boolean algebra are

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

in addition to the axioms of KA above. KAT can model standard imperative programming constructs

$$p\,;q = pq$$
$$\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q = bp + \overline{b}q$$
$$\mathsf{while}\ b\ \mathsf{do}\ p = (bp)^*\overline{b}$$

as well as Hoare partial correctness assertions $\{b\}\,p\,\{c\}$, which can be written in any one of three equivalent ways:

$$bp \leq pc \qquad\qquad bp = bpc \qquad\qquad bp\overline{c} = 0.$$

Hoare-style rules become universal Horn sentences in KAT. For example, the Hoare while-rule

$$\frac{\{bc\}\,p\,\{c\}}{\{c\}\,\mathsf{while}\ b\ \mathsf{do}\ p\,\{\overline{b}c\}}$$

becomes the universal Horn sentence

$$bcp \leq pc \;\Rightarrow\; c(bp)^*\overline{b} \leq (bp)^*\overline{b}\,\overline{b}c.$$

For purposes of program verification, KAT expressions are typically interpreted in binary relation models. Each expression is interpreted as a binary relation on the set of program states, the input/output relation of the program. The tests are interpreted as *subidentities*, subsets of the identity relation on states; a test acts as a guard that either passes the state through unaltered or fails with no output state.

2.3 NetKAT

NetKAT, in its simplest form, is a version of KAT in which the atomic actions and tests take a particular network-specific form, along with some additional axioms for reasoning about programs built using those primitives. The atomic actions are for modifying, duplicating, and forwarding packets, and the atomic tests are for filtering packets based on values of fields.

Formally, the atomic actions and tests are

- $x \leftarrow n$ (assignment)
- $x = n$ (test)
- dup (duplication)

We also use pass and drop for 1 and 0, respectively.

We will describe the formal semantics below, but intuitively, a NetKAT expression is a program that transforms input packets to output packets. The assignment $x \leftarrow n$ assigns the constant value n to the field x in the current

packet. The test $x = n$ tests whether the current value of the field x of the current packet is n and drops the packet if not. For example, the expression

$$\mathit{switch} = 6\,;\mathit{port} = 8\,;\mathit{dest} \leftarrow 10.0.1.5\,;\mathit{port} \leftarrow 5$$

expresses the command: "For all packets incoming on port 8 of switch 6, set the destination IP address to 10.0.1.5 and send the packet out on port 5."

The NetKAT axioms consist of the following equations in addition to the KAT axioms:

$$\begin{aligned}
x \leftarrow n\,;y \leftarrow m &= y \leftarrow m\,;x \leftarrow n \quad (x \neq y) && (2.1)\\
x \leftarrow n\,;y = m &= y = m\,;x \leftarrow n \quad (x \neq y) && (2.2)\\
x = n\,;\mathsf{dup} &= \mathsf{dup}\,;x = n && (2.3)\\
x \leftarrow n\,;x = n &= x \leftarrow n && (2.4)\\
x = n\,;x \leftarrow n &= x = n && (2.5)\\
x \leftarrow n\,;x \leftarrow m &= x \leftarrow m && (2.6)\\
x = n\,;x = m &= 0 \quad (n \neq m) && (2.7)\\
(\textstyle\sum_n x = n) &= 1 && (2.8)
\end{aligned}$$

These equations have the following intuitive interpretations:

(2.1) Assignments to distinct fields may be done in either order.
(2.2) An assignment to a field does not affect the value of a different field.
(2.3) When a packet is duplicated, the field values are preserved.
(2.4) An assignment of a value to a field causes that field to have that value.
(2.5) An assignment to a field of a value that the field already has is redundant.
(2.6) With two assignments to the same field, the second assignment erases the effect of the first.
(2.7) A field may have no more than one value.
(2.8) A field must have at least one value.

2.4 Semantics

The standard model of NetKAT is a packet-forwarding model. Operationally, a NetKAT expression describes a process that maps an input packet to a set of output packets. However, in order to reason about packet trajectories, we need to keep track of changes to the packet as it moves through the network. Thus the standard semantics interprets an expression as a function that maps an input *packet history* to a set of output *packet histories*.

Formally, a *packet* π is a record with constant values n assigned to fields x. A *packet history* is a nonempty sequence of packets

$$\pi_1 :: \pi_2 :: \cdots :: \pi_k.$$

The *head packet* is π_1, which represents the current values of the fields. The remaining packets $\pi_2, \ldots, \pi_k$ describe the previous values from youngest to oldest.

Every NetKAT expression e denotes a function:

$$[\![e]\!] : H \to 2^H$$

where H is the set of all packet histories. The function $[\![e]\!]$ takes an input packet history $\sigma \in H$ and produces a set of output packet histories $[\![e]\!](\sigma) \subseteq H$.

The semantics of expressions is compositional and is defined inductively. For the primitive actions and tests,

$$\begin{aligned}
[\![x \leftarrow n]\!](\pi :: \sigma) &= \{\pi[n/x] :: \sigma\} \\
[\![x = n]\!](\pi :: \sigma) &= \begin{cases} \{\pi :: \sigma\} & \text{if } \pi(x) = n \\ \varnothing & \text{if } \pi(x) \neq n \end{cases} \\
[\![\mathsf{dup}]\!](\pi :: \sigma) &= \{\pi :: \pi :: \sigma\}
\end{aligned}$$

where $\pi[n/x]$ denotes the packet π with the field x rebound to the value n. Thus the assignment $x \leftarrow n$ rebinds the value of x to n in the head packet; the test $x = n$ simply drops the packet (logically, the entire history) if the test is not satisfied and passes it through unaltered if it is satisfied, thus behaving as a packet filter; and dup simply duplicates the head packet. The KAT operations are interpreted as follows:

$$\begin{aligned}
[\![p + q]\!](\sigma) &= [\![p]\!](\sigma) \cup [\![q]\!](\sigma) \\
[\![pq]\!](\sigma) &= \bigcup_{\tau \in [\![p]\!](\sigma)} [\![q]\!](\tau) \\
[\![p^*]\!](\sigma) &= \bigcup_n [\![p^n]\!](\sigma) \\
[\![1]\!](\sigma) &= [\![\mathsf{pass}]\!](\sigma) = \{\sigma\} \\
[\![0]\!](\sigma) &= [\![\mathsf{drop}]\!](\sigma) = \varnothing \\
[\![\neg b]\!](\sigma) &= \begin{cases} \{\sigma\} & \text{if } [\![b]\!](\sigma) = \varnothing \\ \varnothing & \text{if } [\![b]\!](\sigma) = \{\sigma\} \end{cases}
\end{aligned}$$

To compose p and q sequentially, the action p is done first, producing a set of packet histories $[\![p]\!](\sigma)$, then q is performed on each of the resulting histories individually and the results accumulated. This is often called *Kleisli composition*.

The operation $+$ simply accumulates the actions of the two summands. Thus the expression $(\mathit{port} \leftarrow 8) + (\mathit{port} \leftarrow 9)$ describes the behavior of a switch that sends copies of the packet to ports 8 and 9. This is a departure from the usual Kleene interpretation of $+$ as nondeterministic choice—NetKAT treats $+$ as *conjunctive* in the sense that both operations are performed, rather than *disjunctive*, in which one of the two operations would be chosen nondeterministically. Nevertheless, the axioms of NetKAT are sound and complete over this interpretation [1].

3 Examples

In this section we show some useful things that can be done with NetKAT. These examples are all from [1, 10], except some minor improvements have been made in some cases. We will show how various reachability and security properties can be represented as equations between NetKAT terms, thus can be checked automatically by NetKAT's bisimulation-based decision procedure [10]. Specifically, we show how to encode the following queries:

- Reachability: Can host A communicate with host B? Can every host communicate with every other host?
- Security: Does all untrusted traffic pass through the intrusion detection system located at C?
- Loop detection: Is it possible for a packet to be forwarded around a cycle in the network?

Several automated tools already exist for answering such questions [16, 17, 27]. Many of these encode the topology and policy as a logical structure, then translate the query into a Boolean formula and hand it to a SAT solver. In contrast, NetKAT expresses such properties as equations between NetKAT terms, which can then be decided by the NetKAT decision procedure.

3.1 Encoding Network Topology

The topology of the network can be specified by a directed graph with nodes representing hosts and switches and directed edges representing links. In NetKAT, the topology is expressed as a sum of expressions that encode the behavior of each link. To model a link, we use an expression

$$\mathit{switch} = A \,;\, \mathit{port} = n \,;\, \mathit{switch} \leftarrow B \,;\, \mathit{port} \leftarrow m$$

where A and n are the switch name and output port number of the source of the link and B and m are the switch name and input port number of the target of the link. This expression filters out all packets not located at the source end of the link, then updates the switch and port fields to the location of the target of the link, thereby capturing the effect of sending the packet across the link.

3.2 Switch Policies

Each switch may modify and forward packets that it receives on its input ports. The policy for switch A is specified by a NetKAT term

$$\mathit{switch} = A \,;\, p_A$$

where p_A specifies what to do with packets entering switch A. For example, if a packet with IP address a entering on port n should have its IP address modified to b and sent out on ports m and k, this behavior would be expressed by

$$\mathit{port} = n \,;\, \mathit{ip} = a \,;\, (\mathit{port} \leftarrow m + \mathit{port} \leftarrow k) \,;\, \mathit{ip} \leftarrow b$$

and p_A is the sum of all such behaviors for A.

Let t be the sum of all link expressions and p the sum of all switch policies. The product pt describes one step of the network in which each switch processes its packets, then sends them along links to the next switch. Axioms (2.4) and (2.7) guarantee that cross terms in the product vanish, thus the expression correctly captures the linkage. The expression $(pt)^*$ describes the multistep behavior of the network in which the single-step behavior is iterated.

3.3 Reachability

To encode the question of whether it is possible for any packet to travel from an output port of switch A to an input port of switch B given the topology and the switch policies, we can ask whether the expression

$$\mathit{switch} = A \,;\, t(pt)^* \,;\, \mathit{switch} = B \tag{3.1}$$

is equivalent to 0 (drop). Intuitively, the prefix $\mathit{switch} = A$ filters out histories whose head packet does not satisfy $\mathit{switch} = A$, and the postfix $\mathit{switch} = B$ filters out histories whose head packet does not satisfy $\mathit{switch} = B$.

However, more can be said. Using the axioms (2.1)–(2.8), it can be shown that (3.1) is equivalent to a sum of terms of the form

$$\mathit{switch} = A \,;\, x_1 = n_1 \,;\, \cdots \,;\, x_k = n_k \,;\, x_1 \leftarrow m_1 \,;\, \cdots \,;\, x_k \leftarrow m_k \,;\, \mathit{switch} = B$$

and each such nonzero term describes initial conditions under which a packet can travel from A to B. Note that only the initial and final values of the fields appear; the intermediate values vanish due to axioms (2.4), (2.6), and (2.7). We can retain the intermediate values using dup if we wish; an example of this is given below.

3.4 All-Pairs Reachability

We may wish to check whether every host in the network can physically communicate with every other host. To test this, we use the switch policies

$$\mathit{switch} = A \,;\, \sum_n \mathit{port} = n \,;\, \sum_m \mathit{port} \leftarrow m \tag{3.2}$$

where the first sum is over all the active input ports n of node A and the second is over all the active output ports m of A. This expression simply tests whether the packet is currently located at an input port of A and if so forwards it out over all active output ports unaltered. This is a little different from the query of §3.3 in that the switch policies of §3.2, which can modify packets and thus affect traffic flow, are not taken into account, but only the physical network topology.

Let q be the sum of all policies (3.2) over all A. Then q performs this action for all A. Let t be the encoding of the topology as described in §3.1. Consider the equation

$$(qt)^* = \sum_A (\mathit{switch} = A \,;\, \sum_n \mathit{port} = n) \,;\, \sum_B (\mathit{switch} \leftarrow B \,;\, \sum_m \mathit{port} \leftarrow m)$$

where n ranges over all active input ports of A and m ranges over all active input ports of B. The expression qt represents a program that forwards all packets from the input port of any node along all outgoing links to an input port of a node that is reachable in one step. The left-hand expression $(qt)^*$ is the multistep version of this; it starts at an input port of any node A and forwards to all input ports of all nodes reachable from A. The right-hand expression represents a program that, given any packet located a some input port of some node, no matter where it is located, immediately forwards to all input ports of all possible nodes. The left-hand side is contained in the right, since intermediate nodes in a path are elided by axiom (2.6); and if there are A, n, B, m such that input port m of B is not reachable from input port n of A, then

$$\mathit{switch} = A\,;\mathit{port} = n\,;\mathit{switch} \leftarrow B\,;\mathit{port} \leftarrow m$$

will be contained in the right-hand side but not the left.

3.5 Waypointing

A *waypoint* W between A to B is a location that all packets must traverse enroute from A to B. It may be important for security purposes to ensure that all traffic of a certain type traverse a waypoint; for example, we may wish to ensure that all traffic from an untrusted external source to a trusted internal destination traverse a firewall.

We can do this by modifying the switch policy to duplicate the head packet in the firewall component F. That is, the expression $\mathit{switch} = F\,;p_F$ in the sum p is replaced by $\mathit{switch} = F\,;\mathsf{dup}\,;p_F$. This is a way to mark traffic through F. Now we ask whether

$$\begin{aligned}&\mathit{switch} = A\,;t(pt)^*\,;\mathit{switch} = B\\ &\leq \mathit{switch} = A\,;t(pt)^*\,;\mathit{switch} = F\,;\mathsf{dup}\,;p_F\,;t(pt)^*\,;\mathit{switch} = B\end{aligned}$$

which holds if and only if all output packet histories contain a dup generated by traversing F (assuming $F \notin \{A, B\}$).

The solution to this problem presented in [1] inserted a dup in all switch policies; however, the complexity of the decision procedure of [10] is exponential in the number of occurrences of dup, so for performance reasons it is desirable to minimize this quantity. The solution given here has four occurrences.

3.6 Forwarding Loops

A network has a *forwarding loop* if some packet would endlessly traverse a cycle in the network. Forwarding loops are a frequent source of error and have caused outages in both local area networks and on the Internet [15]. They are usually handled by introducing a TTL (time-to-live) field, a runtime mechanism in which a counter is decremented at each hop and the packet is dropped when the counter hits 0.

We can use NetKAT to check for loops by checking whether there is a packet that visits the same state twice. This is done by checking

$$\alpha\,;pt(pt)^*\,;\alpha = 0$$

for each valuation α such that

$$in\,;(pt)^*\,;\alpha$$

does not vanish. Here α represents a valid assignment to all fields and *in* represents a set of initial conditions on packets. The set of α that need to be checked is typically sparse. This algorithm has been used to check for loops in networks with topologies containing thousands of switches and configurations with thousands of forwarding rules on each switch.

3.7 Other Applications

The papers [1, 10] present a few other important applications: traffic isolation, access control, and correctness of a compiler that maps a NetKAT expression to a set of individual flow tables that can be deployed on the switches. It is interesting that so much can be done with regular expressions.

4 Soundness and Completeness

Let $\vdash$ denote provability in ordinary equational logic, assuming the NetKAT axioms (the axioms of KAT plus (2.1)–(2.8)) as premises.

Theorem 1 ([1]). *The NetKAT axioms are sound and complete with respect to the packet-switching semantics of §2.4. That is,* $\vdash p = q$ *if and only if* $[\![p]\!] = [\![q]\!]$.

The completeness proof is quite interesting. It introduces a *language model* for NetKAT that is isomorphic to the packet-switching model of §2.4. The language model also plays a role in the decision procedure of [10]. The language model consists of the regular sets of *reduced strings* of the form

$$\alpha\, p_0\ \mathsf{dup}\ p_1\ \mathsf{dup}\ p_2 \cdots p_{n-1}\ \mathsf{dup}\ p_n, \quad n \geq 0, \tag{4.1}$$

where α is a *complete test* $x_1 = n_1\,;\cdots\,;x_k = n_k$, the p_i are *complete assignments* $x_1 \leftarrow n_1\,;\cdots\,;x_k \leftarrow n_k$, and $x_1, \ldots, x_k$ are all of the fields occurring in the expressions of interest in some arbitrary but fixed order. Every string of atomic actions and tests is equivalent to a reduced string modulo the NetKAT axioms. The set of reduced strings is described by the expression $\mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$, where At is the set of complete tests and P the set of complete assignments. The complete tests are the atoms (minimal nonzero elements) of the Boolean algebra generated by the primitive tests. Complete tests and complete assignments are in one-to-one correspondence as determined by the sequence of values $n_1, \ldots, n_k$.

The standard interpretation over this model is the map G that assigns a regular set of reduced strings to each NetKAT expression:

$$\begin{aligned} G(x \leftarrow n) &= \{\alpha p_\alpha[x \leftarrow n] \mid \alpha \in \mathsf{At}\} \\ G(x = n) &= \{\alpha p_\alpha \mid \alpha \in \mathsf{At},\ x = n \text{ appears in } \alpha\} \\ G(\mathsf{dup}) &= \{\alpha p_\alpha\ \mathsf{dup}\ p_\alpha \mid \alpha \in \mathsf{At}\} \\ G(p + q) &= G(p) \cup G(q) \\ G(pq) &= \{xy \mid \exists \beta\ xp_\beta \in G(p),\ \beta y \in G(q)\} \\ G(p^*) &= \bigcup_{n \geq 0} G(p^n) \end{aligned}$$

where $p[x \leftarrow n]$ denotes the complete assignment p with the assignment to x replaced by $x \leftarrow n$, α_p is the complete test corresponding to the complete assignment p, and p_β is the complete assignment corresponding to the complete test β.

It follows that for $p \in P$ and $\alpha \in \mathsf{At}$,

$$G(p) = \{\alpha p \mid \alpha \in \mathsf{At}\} \qquad\qquad G(\alpha) = \{\alpha p_\alpha\}.$$

The NetKAT axioms (2.1)–(2.8) take a simpler form for reduced strings:

$$\alpha\, \mathsf{dup} = \mathsf{dup}\, \alpha \qquad p\alpha_p = p \qquad \alpha p_\alpha = \alpha$$
$$\alpha\alpha = \alpha \qquad \alpha\beta = 0,\ \alpha \neq \beta \qquad qp = p \qquad \textstyle\sum_{\alpha \in \mathsf{At}} \alpha = 1.$$

5 NetKAT Coalgebra and a Decision Procedure

Coalgebra is a general framework for modeling and reasoning about state-based systems [3, 4, 31, 33, 35]. A central aspect of coalgebra is the characterization of equivalence in terms of *bisimulation*. The bisimulation-based decision procedure for NetKAT presented in [10] was inspired by similar decision procedures for KA and KAT [3, 4, 31]. However, to apply these techniques to NetKAT, it is necessary to develop the coalgebraic theory to provide the basis of the algorithm and establish correctness.

5.1 NetKAT Coalgebra

Formally, a NetKAT coalgebra consists of a set of states S along with *continuation* and *observation maps*

$$\delta_{\alpha\beta} : S \to S \qquad\qquad \varepsilon_{\alpha\beta} : S \to 2$$

for $\alpha, \beta \in \mathsf{At}$. A deterministic NetKAT automaton is a NetKAT coalgebra with a distinguished start state $s \in S$. The inputs to the automaton are the NetKAT

reduced strings (4.1); that is, elements of the set $N = \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$ consisting of strings of the form

$$\alpha p_0 \ \mathsf{dup} \ p_1 \ \mathsf{dup} \cdots \mathsf{dup} \ p_n$$

for some $n \geq 0$. Intuitively, $\delta_{\alpha\beta}$ attempts to consume $\alpha p_\beta \ \mathsf{dup}$ from the front of the input string and move to a new state with a residual input string. This succeeds if and only if the reduced string is of the form $\alpha p_\beta \ \mathsf{dup} \ x$ for some $x \in (P \cdot \mathsf{dup})^* \cdot P$, in which case the automaton moves to a new state as determined by $\delta_{\alpha\beta}$ with residual input string βx. The observation map $\varepsilon_{\alpha\beta}$ determines whether the reduced string αp_β should be accepted in the current state.

Formally, acceptance is determined by a coinductively defined predicate $\mathsf{Accept} : S \times N \to 2$:

$$\begin{aligned} \mathsf{Accept}(t, \alpha p_\beta \ \mathsf{dup} \ x) &= \mathsf{Accept}(\delta_{\alpha\beta}(t), \beta x) \\ \mathsf{Accept}(t, \alpha p_\beta) &= \varepsilon_{\alpha\beta}(t). \end{aligned}$$

A reduced string $x \in N$ is *accepted* by the automaton if $\mathsf{Accept}(s, x)$, where s is the start state.

5.2 The Brzozowski Derivative

The Brzozowski derivative for NetKAT comes in two versions: semantic and syntactic. The semantic version is defined on subsets of N and gives a coalgebra $(2^N, \delta, \varepsilon)$ that is a final coalgebra for the NetKAT signature.

$$\begin{aligned} &\delta_{\alpha\beta} : 2^N \to 2^N && \varepsilon_{\alpha\beta} : 2^N \to 2 \\ &\delta_{\alpha\beta}(A) = \{\beta x \mid \alpha p_\beta \ \mathsf{dup} \ x \in A\} && \varepsilon_{\alpha\beta}(A) = \begin{cases} 1 & \text{if } \alpha p_\beta \in A, \\ 0 & \text{if } \alpha p_\beta \notin A. \end{cases} \end{aligned}$$

One can show that this is the final coalgebra for the NetKAT signature by showing that bisimilarity implies equality.

There is also a syntactic derivative

$$D_{\alpha\beta} : \mathsf{Exp} \to \mathsf{Exp} \qquad\qquad E_{\alpha\beta} : \mathsf{Exp} \to 2,$$

where Exp is the set of reduced NetKAT expressions. The syntactic derivative also gives a coalgebra (Exp, D, E). The maps D and E are defined inductively:

$$D_{\alpha\beta}(p) = 0 \qquad D_{\alpha\beta}(b) = 0 \qquad D_{\alpha\beta}(\mathsf{dup}) = \alpha \cdot \begin{cases} 1 & \text{if } \alpha = \beta, \\ 0 & \text{if } \alpha \neq \beta. \end{cases}$$

$$\begin{aligned} D_{\alpha\beta}(e_1 + e_2) &= D_{\alpha\beta}(e_1) + D_{\alpha\beta}(e_2) \\ D_{\alpha\beta}(e_1 e_2) &= D_{\alpha\beta}(e_1) \cdot e_2 + \sum_\gamma E_{\alpha\gamma}(e_1) \cdot D_{\gamma\beta}(e_2) \\ D_{\alpha\beta}(e^*) &= D_{\alpha\beta}(e) \cdot e^* + \sum_\gamma E_{\alpha\gamma}(e) \cdot D_{\gamma\beta}(e^*) \end{aligned}$$

$$E_{\alpha\beta}(p) = \begin{cases} 1 & \text{if } p = p_\beta, \\ 0 & \text{if } p \neq p_\beta \end{cases} \qquad E_{\alpha\beta}(b) = \begin{cases} 1 & \text{if } \alpha = \beta \leq b, \\ 0 & \text{otherwise} \end{cases}$$

$$E_{\alpha\beta}(\mathsf{dup}) = 0 \qquad E_{\alpha\beta}(e_1 + e_2) = E_{\alpha\beta}(e_1) + E_{\alpha\beta}(e_2)$$

$$E_{\alpha\beta}(e_1 e_2) = \sum_\gamma E_{\alpha\gamma}(e_1) \cdot E_{\gamma\beta}(e_2)$$

$$E_{\alpha\beta}(e^*) = \sum_\gamma E_{\alpha\gamma}(e) \cdot E_{\gamma\beta}(e^*) + \begin{cases} 1 & \text{if } \alpha = \beta, \\ 0 & \text{if } \alpha \neq \beta. \end{cases}$$

Note that the definitions for * are circular, but both are well defined if we take the least fixpoint of the resulting system of equations.

The standard language interpretation $G : \mathsf{Exp} \to 2^N$ is the unique coalgebra morphism to the final coalgebra.

5.3 Matrix Representation

By currying, one can view the signature of NetKAT coalgebra as

$$\delta : X \to X^{\mathsf{At}\times\mathsf{At}} \qquad\qquad \varepsilon : X \to 2^{\mathsf{At}\times\mathsf{At}}$$

and observe that $X^{\mathsf{At}\times\mathsf{At}}$ and $2^{\mathsf{At}\times\mathsf{At}}$ are isomorphic to the families of square matrices over X and 2, respectively, with rows and columns indexed by At. Moreover, as the reader may have noticed, many of the operations used to define the syntactic derivative $D_{\alpha\beta}, E_{\alpha\beta}$ closely resemble matrix operations. Indeed, we can view $\delta(t)$ as an $\mathsf{At} \times \mathsf{At}$ matrix over X and $\varepsilon(t)$ as an $\mathsf{At} \times \mathsf{At}$ matrix over 2. Moreover, if X is a KAT, then the family of $\mathsf{At} \times \mathsf{At}$ matrices over X again forms a KAT, denoted $\mathsf{Mat}(\mathsf{At}, X)$, under the standard matrix operations [7]. Thus we have

$$\delta : X \to \mathsf{Mat}(\mathsf{At}, X) \qquad\qquad \varepsilon : X \to \mathsf{Mat}(\mathsf{At}, 2).$$

With this observation, the syntactic coalgebra defined in §5.2 takes the following concise form:

$$D(p) = 0 \qquad D(b) = 0 \qquad D(\mathsf{dup}) = J \qquad D(e_1 + e_2) = D(e_1) + D(e_2)$$
$$D(e_1 e_2) = D(e_1) \cdot I(e_2) + E(e_1) \cdot D(e_2) \qquad D(e^*) = E(e^*) \cdot D(e) \cdot I(e^*),$$

where $I(e)$ is the diagonal matrix with e on the main diagonal and 0 elsewhere and J is the matrix with α on the main diagonal in position $\alpha\alpha$ and 0 elsewhere; and

$$E(\mathsf{dup}) = 0 \qquad\qquad E(e_1 + e_2) = E(e_1) + E(e_2)$$
$$E(e_1 e_2) = E(e_1) \cdot E(e_2) \qquad\qquad E(e^*) = E(e)^*.$$

In this form E becomes a KAT homomorphism from Exp to $\mathsf{Mat}(\mathsf{At}, 2)$.

Likewise, we can regard the set-theoretic coalgebra presented in §5.2 as having matrix type

$$\delta : 2^N \to \mathsf{Mat}(\mathsf{At}, 2^N) \qquad\qquad \varepsilon : 2^N \to \mathsf{Mat}(\mathsf{At}, 2).$$

Again, in this form, ε becomes a KAT homomorphism.

This matrix representation is exploited heavily in the implementation of the decision procedure of [10] described below in §6.

5.4 Kleene's Theorem for NetKAT

The correctness of the bisimulation algorithm hinges on the relationship between the coalgebras described in §5.1 and the packet-switching and language models described in §2.4 and §4, respectively. This result is the generalization to NetKAT of Kleene's theorem relating regular expressions and finite automata.

Theorem 2 ([10]). *A set of NetKAT reduced strings is the set accepted by some finite-state NetKAT automaton if and only if it is $G(e)$ for some NetKAT expression e.*

Given a NetKAT expression e, an equivalent finite NetKAT automaton can be constructed from the derivatives of e modulo associativity, commutativity, and idempotence (ACI), with e as the start state. The continuation and observation maps are the syntactic derivative introduced in §5.2. A careful analysis shows that the number of states is at most $|\mathsf{At}|\cdot 2^\ell$, where ℓ is the number of occurrences of dup in e.

6 Implementation

The paper [10] describes an implementation of the decision procedure for NetKAT term equivalence. It converts two NetKAT terms to automata using Brzozowski derivatives, then tests bisimilarity. The implementation comprises roughly 4500 lines of OCaml and includes a parser, pretty printer, and visualizer. The implementation has been integrated into the Frenetic SDN controller platform and has been tested on numerous benchmarks with good results.

The bisimilarity algorithm is fairly standard. Given two NetKAT terms, all derivatives are calculated, and the E matrices of corresponding pairs are checked for equality. The procedure fails immediately if they are not. This coinductive algorithm can be implemented in almost linear time in the combined size of the automata using the union-find data structure of Hopcroft and Karp [13] to represent the bisimilarity classes.

6.1 Optimizations

The implementation incorporates a number of important enhancements and optimizations to avoid combinatorial blowup. It uses a symbolic representation that

exploits symmetry and sparseness to reduce the size of the state space. Intermediate values that do not contribute to the final outcome are aggressively pruned. To further improve performance, the implementation incorporates a number of other optimizations: hash-consing and memoization, sparse multiplication, base compaction, fast computaton of fixpoints. These enhancements are described in detail in [10].

Although the algorithm is still necessarily exponential in the worst case (the problem is PSPACE-complete), the tool tends to be fast in practice due to the constrained nature of real-world problems.

7 Related Work

Software-defined networking (SDN) has emerged in recent years as the dominant paradigm for network programming. A number of SDN programming languages and verification tools have appeared [2,5,8,9,12,16–18,26,28–30,34,36–39], and SDN is being actively deployed in industry [14,20,21].

NetKAT [1, 10] was developed as a part of the Frenetic project [9, 12, 28, 29]. Compared to other tools, NetKAT is unique in its focus on algebraic and coalgebraic structure of network programs. NetKAT largely inherits its syntax, semantics, and application methodology from these earlier efforts but adds a complete deductive system and PSPACE decision procedure.

The algebraic and coalgebraic theories of KA and KAT and related systems have been studied extensively [6, 22–25, 33, 35]. This work has uncovered strong relationships between the algebraic/logical view of systems and the combinatorial/automata-theoretic view. These ideas have figured prominently in the development of NetKAT.

The implementation uses many ideas and optimizations from the coalgebraic implementations of KA and KAT and other related systems [3,4,31] to provide enhanced performance, making automated decision feasible even in the face of PSPACE completeness.

8 Conclusion

This paper surveys recent work on NetKAT, a relatively new language and logic for specifying and verifying network packet-processing functions. NetKAT was introduced in [1] and further developed in [10]. We have attempted to make the presentation self-contained and accessible, but a more comprehensive treatment can be found in the original papers.

NetKAT consists of Kleene algebra with tests [23] with specialized primitives for expressing properties of networks, along with equational axioms for reasoning with those constructs. The standard semantics is a packet-switching model that interprets NetKAT expressions as functions from packet histories to sets of packet histories. There is also a language model that is isomorphic to the packet-switching model and a coalgebraic model that is related to the other two models

via a version of Kleene's theorem. The NetKAT axioms are sound and complete over these interpretations. The coalgebraic model admits a bisimulation-based decision procedure that is efficient in many cases of practical interest, although the general problem is PSPACE-complete. There is a full implementation in OCaml that is efficient in practice and compares favorably with the state of the art. Several applications of interest have also been described.

Acknowledgments

Special thanks to my coauthors Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Matthew Milano, Cole Schlesinger, Alexandra Silva, Laure Thompson, and David Walker for their kind permission to include results from [1, 10] in this survey. Thanks also to Konstantinos Mamouras, Andrew Myers, Mark Reitblatt, Ross Tate, and the rest of the Cornell PLDG group for many insightful discussions and helpful comments.

References

1. Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL'14)*, pages 113–126, San Diego, California, USA, January 2014. ACM.
2. Thomas Ball, Nikolaj Bjorner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *PLDI*, 2014. To appear.
3. Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *Proc. 40th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, POPL '13, pages 457–468. ACM, 2013.
4. Thomas Braibant and Damien Pous. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, 8(1:16):1–42, 2012.
5. Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.
6. Hubie Chen and Riccardo Pucella. A coalgebraic approach to Kleene algebra with tests. *Electronic Notes in Theoretical Computer Science*, 82(1), 2003.
7. Ernie Cohen, Dexter Kozen, and Frederick Smith. The complexity of Kleene algebra with tests. Technical Report TR96-1598, Computer Science Department, Cornell University, July 1996.
8. Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *SIGCOMM*, 2013.
9. Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, September 2011.

10. Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. Technical Report `http://hdl.handle.net/1813/36255`, Computing and Information Science, Cornell University, March 2014. POPL 2015, to appear.
11. Open Networking Foundation. Software-defined networking: The new norm for networks. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf`, 2012. White paper.
12. Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, June 2013.
13. John E. Hopcroft and Richard M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California, 1971.
14. Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.
15. Ethan Katz-Bassett, Colin Scott, David R. Choffnes, Ítalo Cunha, Vytautas Valancius, Nick Feamster, Harsha V. Madhyastha, Thomas Anderson, and Arvind Krishnamurthy. Lifeguard: Practical repair of persistent route failures. In *SIGCOMM '12*, 2012.
16. Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.
17. Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.
18. Hyojoon Kim and N. Feamster. Improving network management with software defined networking. *Communications Magazine, IEEE*, 51(2):114–119, February 2013.
19. Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, N.J., 1956.
20. Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.
21. Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.
22. Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.
23. Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.
24. Dexter Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report `http://hdl.handle.net/1813/10173`, Computing and Information Science, Cornell University, March 2008.
25. Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.

26. Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, 2005.
27. Haohui Mai, Ahmed Khurshid, Raghit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.
28. Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, January 2012.
29. Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *NSDI*, April 2013.
30. Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *HotSDN*, 2013.
31. Damien Pous. Relational algebra and KAT in Coq, February 2013. Available at `http://perso.ens-lyon.fr/damien.pous/ra`.
32. J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.
33. Jan J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer, 1998.
34. Robert Colin Scott, Andreas Wundsam, Kyriakos Zarifis, and Scott Shenker. What, Where, and When: Software Fault Localization for SDN. Technical Report UCB/EECS-2012-178, EECS Department, University of California, Berkeley, 2012.
35. Alexandra Silva. *Kleene Coalgebra*. PhD thesis, University of Nijmegen, 2010.
36. Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, 2011.
37. Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.
38. Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.
39. Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *CoNEXT*, 2012.