A Coalgebraic Decision Procedure for NetKAT

Nate Foster
Cornell University

Dexter Kozen
Cornell University

Mae Milano
Cornell University

Alexandra Silva
Radboud University Nijmegen

Laure Thompson
Cornell University

Abstract

NetKAT is a domain-specific language and logic for specifying and verifying network packet-processing functions. It consists of Kleene algebra with tests (KAT) augmented with primitives for testing and modifying packet headers and encoding network topologies. Previous work developed the design of the language and its standard semantics, proved the soundness and completeness of the logic, defined a PSPACE algorithm for deciding equivalence, and presented several practical applications.

This paper develops the coalgebraic theory of NetKAT, including a specialized version of the Brzozowski derivative, and presents a new efficient algorithm for deciding the equational theory using bisimulation. The coalgebraic structure admits an efficient sparse representation that results in a significant reduction in the size of the state space. We discuss the details of our implementation and optimizations that exploit NetKAT's equational axioms and coalgebraic structure to yield significantly improved performance. We present results from experiments demonstrating that our tool is competitive with state-of-the-art tools on several benchmarks including all-pairs connectivity, loop-freedom, and translation validation.

Categories and Subject Descriptors F.4.3 [*Formal Languages*]: Classes defined by grammars or automata

Keywords Coalgebra; Kleene algebra with tests; Brzozowski derivatives; automata; network verification; NetKAT.

1. Introduction

Networks have received widespread attention in recent years as a target for domain-specific language design. The emergence of *software-defined networking* (SDN) as a popular paradigm for network programming has led to the appearance of a number of SDN programming languages including Frenetic, Nettle, NetCore, Pyretic, Maple, and PANE, among others [10–12, 26, 27, 39, 40]. The details of these languages differ, but each seeks to provide high-level abstractions to simplify the task of specifying the packet-processing behavior of a network. In addition to SDN languages, a number of verification tools including HSA, VeriFlow, FlowLog, and VeriCon are also being actively developed [2, 16, 17, 28]. As SDN is being deployed in production enterprise, data center, and wide-area networks [14, 19, 20], it is becoming clear that SDN is the next major step in the evolution of network technology and is destined to have a significant impact.

Previous work by Anderson et al. [1] introduced NetKAT, a language and logic for specifying and verifying the packet-processing behavior of networks. NetKAT provides general-purpose programming constructs such as parallel and sequential composition, conditional tests, and iteration, as well as special-purpose primitives for querying and modifying packet headers and encoding network topologies. The language allows the desired behavior of a network to be specified equationally. In contrast to competing approaches, NetKAT has a formal mathematical semantics and an equational deductive system that is sound and complete over that semantics, as well as a PSPACE decision procedure. It is based on Kleene algebra with tests (KAT), an algebraic system for propositional program verification that has been extensively studied for nearly two decades [22]. Several practical applications of NetKAT have been developed, including algorithms for testing reachability and non-interference and a syntactic correctness proof for a compiler that translates programs to hardware instructions for SDN switches.

This paper develops the coalgebraic theory of NetKAT, defines a new algorithm for deciding equivalence based on this technology, and presents a full implementation in OCaml. The new algorithm is significantly more efficient than the previous naive algorithm [1], which was PSPACE in the best case and the worst case, as it was based on the determinization of a nondeterministic algorithm.

The contributions of this paper are both theoretical and practical. On the theoretical side, we introduce a new coalgebraic model of NetKAT, including a specialized version of the Brzozowski derivative in both semantic and syntactic forms. We prove a version of Kleene's theorem for NetKAT that shows that the coalgebraic model is equivalent to the standard packet-processing and language models introduced previously [1]. A highlight of our theoretical development is a representation theorem showing that the Brzozowski derivative can be concisely encoded in matrix form. On the practical side, we develop a new coalgebraic decision procedure for term equivalence based on our theoretical results, along with a full implementation in OCaml. The algorithm constructs a bisimulation between coalgebras built from NetKAT expressions via the Brzozowski derivative. The matrix representation enables us to exploit sparseness to obtain a significant reduction in the size of the state space. The implementation is very efficient in practice—it can verify reachability in a real-world campus network in less than a

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/10.1145/2676726.2677011

second on a laptop. We demonstrate the real-world applicability of our tool by using it to decide common network verification questions such as all-pairs connectivity, loop-freedom, and translation validation—all pressing questions in modern networks. The results of experiments on these benchmarks demonstrates that our implementation compares favorably with the state of the art.

The rest of this paper is organized as follows. In §2 we briefly review the syntax and semantics of NetKAT [1]. In §3 we introduce NetKAT coalgebras along with a variant of the Brzozowski derivative. In §4 we prove our main theoretical result on which the correctness of our equivalence algorithm is based: a generalization of Kleene's theorem relating NetKAT expressions and NetKAT coalgebras. In §5 we discuss a streamlined representation of NetKAT coalgebras using matrices, which is needed for our implementation. In §6 we present the details of our implementation, focusing on how we exploit the NetKAT axioms and coalgebraic structure to achieve significant performance improvements over the naive algorithm defined previously [1]. In §7 we describe three applications developed from our coalgebraic theory, which are used in the evaluation of our implementation. In §8 we report on the results of experiments. In §9 we discuss related work, and in §10 we present conclusions and identify directions for future research.

2. Overview

In this section we briefly review the syntax and semantics of NetKAT, along with other results that are needed to understand our coalgebraic algorithm described in the following sections [1].

NetKAT is based on Kleene algebra with tests (KAT) [22], a generic equational system for reasoning about partial correctness of programs. KAT is Kleene algebra (KA), the algebra of regular expressions, augmented with Boolean tests. Formally, a 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 Kleene algebra operators are choice ($+$); sequential composition ($\cdot$), which is often elided in expressions; iteration (*); fail (0), and skip (1). Elements of B are called *tests*. On tests, choice, sequential composition behave as Boolean disjunction and conjunction, respectively, and 0 and 1 stand for falsity and truth, respectively. The operator $\bar{}$ is the Boolean negation operator, sometimes written as $\neg$. The axioms of Kleene algebra are as follows:

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

where $p \leq q \Leftrightarrow p + q = q$. The axioms of Boolean algebra are

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

in addition to the axioms of Kleene algebra above. KAT can model standard imperative programming constructs,

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

as well as Hoare partial correctness assertions

$$\{b\}\,p\,\{c\} \;\Leftrightarrow\; bp \leq pc \;\Leftrightarrow\; bp = bpc \;\Leftrightarrow\; bp\overline{c} = 0.$$

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

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

KA and KAT have standard language models consisting of, respectively, the regular sets of finite-length strings over a finite alphabet and the regular sets of *guarded strings* over disjoint finite alphabets of test and action symbols. These language models play an important role in that they are the free models on their generators, which means that they exactly characterize the equational theory. There are other useful models, including binary relation and trace models used in programming language semantics. KAT is complete for the equational theory of binary relation models. The equational theories of KA and KAT are both PSPACE complete.

NetKAT extends KAT with network-specific primitives for filtering, modifying, and forwarding packets, along with additional axioms for reasoning about programs built using those primitives. More formally, NetKAT is KAT with primitive actions and tests

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

We also use id and drop for 1 and 0, respectively. Intuitively, the assignment $x \leftarrow n$ assigns the value n to the field x in the current packet. The test $x = n$ tests whether field x of the current packet contains the value n. The action dup duplicates the packet in the packet history, which keeps track of the path the packet takes through the network. As an example, the expression

$$switch = 6\,;\, port = 8\,;\, ipDst \leftarrow 10.0.1.5\,;\, port \leftarrow 5$$

encodes the command: "For all packets located at port 8 of switch 6, set the destination IP address to 10.0.1.5 and forward the packet out on port 5."

The NetKAT axioms consist of the KAT axioms as well as the following axioms, which govern the behavior of tests, assignments, duplication, and the interactions between them:

$$\begin{array}{rcl} x \leftarrow n\,;y \leftarrow m & = & y \leftarrow m\,;x \leftarrow n \quad (\text{if } x \neq y) \\ x \leftarrow n\,;y = m & = & y = m\,;x \leftarrow n \quad (\text{if } x \neq y) \\ x = n\,;\mathsf{dup} & = & \mathsf{dup}\,;x = n \\ x \leftarrow n\,;x = n & = & x \leftarrow n \\ x = n\,;x \leftarrow n & = & x = n \\ x \leftarrow n\,;x \leftarrow m & = & x \leftarrow m \\ x = n\,;x = m & = & 0 \quad (\text{if } n \neq m) \\ \left(\sum_n x = n\right) & = & 1 \end{array}$$

Intuitively, the first axiom states that assignments to distinct fields may be done in either order. The third axiom says that when a packet is duplicated, the values of the fields in the head packet are preserved in the history. The other axioms have similar intuitive interpretations.

There are many models that satisfy the NetKAT axioms, but the standard model of NetKAT is formulated in terms of packet-processing functions. A *packet* π is a record whose fields assign constant values n to fields f. A *packet history* is a nonempty

sequence of packets

$$\pi_1 :: \pi_2 :: \cdots :: \pi_k,$$

in which the *head packet* is π_1. Operationally, only the head packet exists in the network, but in the logic we keep track of the packet's history to enable precise specification of forwarding behavior involving specific paths through the network. Every NetKAT expression e denotes a function:

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

where H is the set of all packet histories. Intuitively, the expression e takes an input packet history σ and produces a set of output packet histories $[\![e]\!](\sigma)$.

The semantics of the primitive actions and tests are as follows. For a packet history $\pi :: \sigma$ with head packet π,

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

where $\pi[n/x]$ denotes packet π with the field x rebound to the value n. Note that a test $x = n$ drops the packet if the test is not satisfied and passes it through unaltered if it is satisfied—i.e., tests behave as filters on packets. The dup construct duplicates the head packet π, yielding a fresh copy that can be modified by other constructs. Hence, in this standard model, the dup construct can be used to encode paths through the network, with each occurrence of dup marking an intermediate hop.

The KAT operations are interpreted as follows:

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

The interpretation of sequential composition is often called *Kleisli composition*, as it is composition in the Kleisli category of the powerset monad. The $+$ operator accumulates actions. Thus the expression $(port \leftarrow 8) + (port \leftarrow 9)$ describes the behavior of a switch that outputs a copy of the packet on ports 8 and 9. Note that this is a departure from the usual Kleene algebra interpretation of $+$ as nondeterministic choice—NetKAT treats it as conjuctive rather than disjunctive. Nevertheless, it is not difficult to show that the axioms of KAT and NetKAT are sound over this interpretation.

The proof of completeness is more difficult, and uses a language model that plays a similar role as the regular sets of strings do for KA and the regular sets of guarded strings do for KAT [1]. The language model for NetKAT consists of the regular sets of *reduced strings* of the form

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

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 in some arbitrary but fixed order.[1] Every NetKAT expression can be rewritten to an equivalent *reduced expression* in which every test is a complete test and every assignment is a complete assignment. Likewise, every string of primitive 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 determined by the sequence of values $n_1, \ldots, n_k$.

It is straightforward to show that every NetKAT expression e can be interpreted as a regular set of reduced strings $G(e)$. The NetKAT axioms can be expressed in a simpler form for reduced strings. Let α_p be the complete test corresponding to the complete assignment p. Likewise, let p_β be the complete assignment corresponding to the complete test β. The NetKAT axioms for reduced strings are as follows:

$$\alpha\, \mathsf{dup} = \mathsf{dup}\, \alpha \qquad p\alpha_p = p \qquad \alpha p_\alpha = \alpha$$

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

See the previous paper on NetKAT [1] for a comprehensive treatment of the language model, including proofs of the claims above.

3. NetKAT Coalgebra

Coalgebra is a general framework for modeling and reasoning about state-based systems [4, 5, 30, 33, 36]. A central aspect of coalgebra is the characterization of equivalence in terms of *bisimulation*. Our work is motivated by recent experiences with bisimulation-based decision procedures for KA and KAT [4, 5, 30]. However, to apply these techniques to NetKAT, we must first develop its coalgebraic theory. This will provide a combinatorial view of NetKAT similar to classical automata theory for KA and automata on guarded strings for KAT. This section develops this theory, which provides the necessary structure for our bisimulation-based decision procedure.

For background on the general theory of coalgebra in modeling state-based systems, see the survey article by Rutten [34]. The only general knowledge needed from this domain is the following:

1. Coalgebras are usually defined in terms of a set of *states* along with *observation* and *continuation* maps. The observation map gives information about each state while the continuation map specifies transition(s) from one state to the next states(s). The nature of these maps varies depending on the type of the system being modeled.
2. Two states are considered *bisimilar* if the observation maps yield identical information for both states and the continuation map leads again to bisimilar states.
3. A *homomorphism* is a map between coalgebras that preserves the structure of observations and continuations.
4. There is often a *final coalgebra* into which there is a unique homomorphism from any other coalgebra of the same type. Two states are bisimilar if and only if this homomorphism maps them to the same state in the final coalgebra.

As an example, a deterministic automaton over a finite alphabet Σ is a coalgebra with an observation map $S \to 2$ that indicates whether a state is an accepting state and a continuation map $S \times \Sigma \to S$ that specifies the transitions of the automaton. The final coalgebra is 2^{Σ^*}, the powerset of the set of all strings over Σ, with observation and continuation maps given by the (semantic) Brzozowski derivative $\varepsilon : 2^{\Sigma^*} \to 2$ such that $\varepsilon(L) = 1$ if and only if L contains the null string and $\delta(L, a) = \{w \mid aw \in L\}$ for $a \in \Sigma$ and $w \in \Sigma^*$. The unique homomorphism from an

[1] Note that we will use metavariables p to range over NetKAT expressions as well as complete tests. The intended meaning will be clear from context.

automaton to the final coalgebra takes a state s to the set of strings that would be accepted by the automaton if s were the start state.

There is also a *syntactic* Brzozowski derivative defined inductively on regular expressions Exp over Σ:

$$D_a(e_1{+}e_2) = D_a(e_1) + D_a(e_2)$$
$$D_a(e_1e_2) = D_a(e_1)\cdot e_2 + E(e_1)\cdot D_a(e_2)$$
$$D_a(e^*) = D_a(e)\cdot e^*$$
$$D_a(1) = D_a(0) = 0 \qquad D_a(b) = [b = a]$$

$$E(e_1{+}e_2) = E(e_1) + E(e_2)$$
$$E(e_1e_2) = E(e_1)\cdot E(e_2)$$
$$E(e^*) = 1$$
$$E(1) = 1 \qquad E(0) = E(a) = 0,\ a \in \Sigma,$$

where $[\varphi]$ = 1 or 0 according as φ is true or false, respectively. The map taking a regular expression e to the set of strings it represents is the unique homomorphism to the final coalgebra.

3.1 Definitions

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

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

for $\alpha, \beta \in \mathsf{At}$. A deterministic NetKAT automaton is simply a finite-state NetKAT coalgebra with a distinguished start state $s \in S$. (There are also corresponding notions of nondeterministic automaton and a determinization procedure, but we will not need these for our formal development in this paper.) The inputs to the automaton are reduced strings belonging to the set $U = \mathsf{At}\cdot P\cdot(\mathsf{dup}\cdot P)^*$. That is, U contains 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 string αp_β should be accepted in the current state.

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

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

A reduced string $x \in U$ is *accepted* by the automaton if $\mathsf{Accept}(s, x)$, where s is the start state. A NetKAT coalgebra is a coalgebra for the set endofunctor

$$FX = X^{\mathsf{At}\times\mathsf{At}} \times 2^{\mathsf{At}\times\mathsf{At}} \tag{3.1}$$

The continuation and observation maps comprise the structure map of the coalgebra:

$$(\delta, \varepsilon) : X \to FX.$$

One can see immediately from equation (3.1) 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. Indeed, in §5, we will exploit the one-to-one correspondence between P and At to express δ and ε in matrix form.

The reader familiar with coalgebra might notice that the final coalgebra of the above functor is not exactly reduced NetKAT strings. However, the semantics of a NetKAT automaton as acceptor of reduced NetKAT strings can be recovered coalgebraically by doing a generalized powerset construction [38] in which one of the atoms in the argument of δ is hidden in the state. This is analogous to the situation for nondeterministic finite automata: these are (compact) acceptors of languages which need to be made deterministic in order to recover language semantics as the canonical equivalence. For space reasons and to keep the presentation simple, we will not explain the generalized powerset construction involved in recovering the language semantics categorically, but rather give the concrete definitions of the semantic map and syntactic structure on expressions.

3.2 The Brzozowski Derivative

This section develops a variant of the Brzozowski derivative for NetKAT. The derivative comes in two versions: semantic and syntactic. The semantic version is defined on subsets of U and gives rise to a NetKAT coalgebra $(2^U, \delta, \varepsilon)$. The syntactic version is defined on expressions and also gives rise to a coalgebra (Exp, D, E). There is a unique language interpretation $G : \mathsf{Exp} \to 2^U$.

Language Semantics. The language semantics for NetKAT is given by the semantic derivative:

$$\delta_{\alpha\beta} : 2^U \to 2^U \qquad \varepsilon_{\alpha\beta} : 2^U \to 2$$
$$\delta_{\alpha\beta}(A) = \{\beta x \mid \alpha p_\beta\ \mathsf{dup}\ x \in A\} \qquad \varepsilon_{\alpha\beta}(A) = [\alpha p_\beta \in A].$$

Syntactic Coalgebra. There is also a syntactic derivative:

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

where Exp is the set of reduced NetKAT expressions.[2] It is defined inductively as follows:

$$D_{\alpha\beta}(p) = 0 \qquad D_{\alpha\beta}(b) = 0 \qquad D_{\alpha\beta}(\mathsf{dup}) = \alpha\cdot[\alpha = \beta]$$
$$D_{\alpha\beta}(e_1 + e_2) = D_{\alpha\beta}(e_1) + D_{\alpha\beta}(e_2)$$
$$D_{\alpha\beta}(e_1e_2) = D_{\alpha\beta}(e_1)\cdot e_2 + \sum_\gamma E_{\alpha\gamma}(e_1)\cdot D_{\gamma\beta}(e_2)$$
$$D_{\alpha\beta}(e^*) = D_{\alpha\beta}(e)\cdot e^* + \sum_\gamma E_{\alpha\gamma}(e)\cdot D_{\gamma\beta}(e^*)$$

$$E_{\alpha\beta}(p) = [p = p_\beta] \qquad E_{\alpha\beta}(b) = [\alpha = \beta \leq b]$$
$$E_{\alpha\beta}(\mathsf{dup}) = 0 \qquad E_{\alpha\beta}(e_1 + e_2) = E_{\alpha\beta}(e_1) + E_{\alpha\beta}(e_2)$$
$$E_{\alpha\beta}(e_1e_2) = \sum_\gamma E_{\alpha\gamma}(e_1)\cdot E_{\gamma\beta}(e_2)$$
$$E_{\alpha\beta}(e^*) = [\alpha = \beta] + \sum_\gamma E_{\alpha\gamma}(e)\cdot E_{\gamma\beta}(e^*).$$

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

4. Kleene's Theorem for NetKAT

In this section we prove that a subset of U is $G(e)$ for some NetKAT expression e if and only if it is the set of strings accepted by some finite NetKAT automaton. This result is the generalization of Kleene's theorem, which relates regular expressions and automata, to NetKAT.

[2] Readers familiar with previous work on NetKAT [1] may notice that the syntactic derivative is actually defined on a superset of reduced NetKAT expressions that includes arbitrary tests b. The definition given here illustrates the connection to previous work on derivatives in the context of KAT [23] and remains correct when restricted to complete tests.

4.1 From Automata to Expressions

Let $M = (S, \delta, \varepsilon, s)$ be a finite NetKAT automaton. Consider a graph H with nodes $(S \times \mathsf{At}) \cup \{\mathsf{halt}\}$ and labeled edges

$$(u,\alpha) \xrightarrow{p_\beta \mathsf{dup}} (v,\beta), \qquad \text{if } \delta_{\alpha\beta}(u) = v$$

$$(u,\alpha) \xrightarrow{p_\beta} \mathsf{halt}, \qquad \text{if } \varepsilon_{\alpha\beta}(u) = 1.$$

We claim that for $x \in (P \cdot \mathsf{dup})^* \cdot P$,

$$(t,\alpha) \xrightarrow{x} \mathsf{halt} \;\Leftrightarrow\; \mathsf{Accept}(t, \alpha x). \tag{4.1}$$

This can be proved by induction on the length of x. For the basis,

$$(t,\alpha) \xrightarrow{p_\beta} \mathsf{halt} \;\Leftrightarrow\; \varepsilon_{\alpha\beta}(t) = 1 \;\Leftrightarrow\; \mathsf{Accept}(t, \alpha p_\beta).$$

For the induction step,

$$\begin{aligned}(t,\alpha) \xrightarrow{p_\beta \mathsf{dup} x} \mathsf{halt} &\;\Leftrightarrow\; \exists u\, (t,\alpha) \xrightarrow{p_\beta \mathsf{dup}} (u,\beta) \xrightarrow{x} \mathsf{halt} \\ &\;\Leftrightarrow\; \exists u\, \delta_{\alpha\beta}(t) = u \wedge \mathsf{Accept}(u, \beta x) \\ &\;\Leftrightarrow\; \mathsf{Accept}(\delta_{\alpha\beta}(t), \beta x) \\ &\;\Leftrightarrow\; \mathsf{Accept}(t, \alpha p_\beta\ \mathsf{dup}\ x).\end{aligned}$$

The set of labels of paths in H from (t,α) to halt is a regular subset of $(P \cdot \mathsf{dup})^* \cdot P$ and is described by a regular expression $e(t,\alpha)$. These expressions can be computed by taking the star of H considered as a square matrix. By (4.1), the set of strings accepted by M is the regular subset of U described by $e = \sum_\alpha \alpha \cdot e(s,\alpha)$.

As shown previously [1], if $R(e) \subseteq U$, where R is the canonical interpretation of regular expressions as regular sets of strings, then $R(e) = G(e)$. Hence, we have the following theorem.

Theorem 1. *Let M be a finite NetKAT automaton. There exists a NetKAT expression e such that the set of reduced strings accepted by M is $G(e)$.*

4.2 From Expressions to Automata

For the other direction, we show how to construct a finite NetKAT automaton M_e from an expression e. The states of the automaton are NetKAT expressions modulo associativity, commutativity, and idempotence (ACI), with e as the start state. The continuation and observation maps are the syntactic derivative introduced in §3.2.

Lemma 1. *The set accepted by M_e is $G(e)$.*

Proof. By Lemma 4, G is a coalgebra homomorphism from the syntactic coalagebra (Exp, D, E) to the set-theoretic coalgebra $(2^U, \delta, \varepsilon)$. Proceeding by induction on the length of the string, we have the following:

$$\begin{aligned}\mathsf{Accept}(e, \alpha p_\beta) &\Leftrightarrow E_{\alpha\beta}(e) = 1 \\ &\Leftrightarrow G(E_{\alpha\beta}(e)) = 1 \\ &\Leftrightarrow \varepsilon_{\alpha\beta}(G(e)) = 1 \\ &\Leftrightarrow \alpha p_\beta \in G(e),\end{aligned}$$

$$\begin{aligned}\mathsf{Accept}(e, \alpha p_\beta\ \mathsf{dup}\ x) &\Leftrightarrow \mathsf{Accept}(D_{\alpha\beta}(e), \beta x) \\ &\Leftrightarrow \beta x \in G(D_{\alpha\beta}(e)) \\ &\Leftrightarrow \beta x \in \delta_{\alpha\beta}(G(e)) \\ &\Leftrightarrow \alpha p_\beta\ \mathsf{dup}\ x \in G(e).\end{aligned}$$

□

It remains to show that M_e is finite. This follows from the fact that e has finitely many derivatives up to ACI. We defer the proof of this fact to Lemma 6 in the next section, as it depends on some details of our data representation.

Theorem 2. *For every NetKAT expression e, there is a deterministic NetKAT automaton M_e with at most $|\mathsf{At}| \cdot 2^\ell$ states accepting the set $G(e)$, where ℓ is the number of occurrences of* dup *in e.*

5. Term and Automata Representations

In this section, we develop a collection of concrete structures that are useful for representing NetKAT automata and will lead to a practical implementation. They also provide further theoretical insights into the structure of the NetKAT language.

5.1 Matrices

The reader has probably noticed that many of the operations used to define the syntactic derivative $D_{\alpha\beta}$ and $E_{\alpha\beta}$ closely resemble matrix operations. Indeed, if we regard the types of the coalgebra operations as having the following types:

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

then 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 [9]. Thus we have

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

So the syntactic coalgebra defined in §3.2 takes the following form:

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

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. Similarly, we have:[3]

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

Note that 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 §3.2 as having type:

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

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

Lemma 2.

(i) $\varepsilon(1) = I$
(ii) $\varepsilon(A \cup B) = \varepsilon(A) + \varepsilon(B)$
(iii) $\varepsilon(A \cdot B) = \varepsilon(A) \cdot \varepsilon(B)$
(iv) $\varepsilon(A^*) = \varepsilon(A)^*$

Proof. These properties follow straightforwardly from the definitions in §3.2. For example, for (iii) and (iv), we have

$$\begin{aligned}\varepsilon(AB)_{\alpha\beta} &= [\alpha p_\beta \in AB] \\ &= [\exists\gamma\ \alpha p_\gamma \in A \wedge \gamma p_\beta \in B] \\ &= \sum_\gamma [\alpha p_\gamma \in A] \cdot [\gamma p_\beta \in B] \\ &= \sum_\gamma \varepsilon(A)_{\alpha\gamma} \cdot \varepsilon(B)_{\gamma\beta} \\ &= (\varepsilon(A) \cdot \varepsilon(B))_{\alpha\beta}\end{aligned}$$

$$\varepsilon(A^*) = \varepsilon(\textstyle\bigcup_n A^n) = \sum_n \varepsilon(A)^n = \varepsilon(A)^*.$$

□

[3] Here, we elide the cases for tests b and complete assignments p.

The next lemma characterizes δ on the regular operators.

Lemma 3.

(i) $\delta(\bigcup_n A_n) = \sum_n \delta(A_n)$
(ii) $\delta(AB) = \delta(A) \cdot I(B) + \varepsilon(A) \cdot \delta(B)$
(iii) $\delta(A^*) = \varepsilon(A^*) \cdot \delta(A) \cdot I(A^*)$

where $I(A)$ is the matrix with the set A on the main diagonal and $\varnothing$ elsewhere, and the matrix sum in (i) is componentwise union.

Proof. We argue (ii) and (iii) explicitly; (i) follows from linearity.

(ii) By definition, $\delta_{\alpha\beta}(AB) = \{\beta x \mid \alpha p_\beta \ \mathsf{dup}\ x \in AB\}$. To show that $\alpha p_\beta \ \mathsf{dup}\ x \in AB$, the string must be the product of two reduced strings, one from A and one from B. Depending on which of these strings contains the first occurrence of dup, one of the following must occur: (1) there exists γ such that $\alpha p_\beta \ \mathsf{dup}\ x = \alpha p_\gamma \cdot \gamma p_\beta \ \mathsf{dup}\ x$ with $\alpha p_\gamma \in A$ and $\gamma p_\beta \ \mathsf{dup}\ x \in B$; or (2) there exist γ, y, and z such that $\alpha p_\beta \ \mathsf{dup}\ x = \alpha p_\beta \ \mathsf{dup}\ y p_\gamma \cdot \gamma z$ with $\alpha p_\beta \ \mathsf{dup}\ y p_\gamma \in A$, $\gamma z \in B$, and $x = y p_\gamma \gamma z$.

In the first case, we have $\varepsilon_{\alpha\gamma}(A) = 1$ and $\beta x \in \delta_{\gamma\beta}(B)$, hence $\beta x \in \varepsilon_{\alpha\gamma}(A) \cdot \delta_{\gamma\beta}(B)$. In the second case, we have $\beta y p_\gamma \in \delta_{\alpha\beta}(A)$ and $\gamma z \in B$, hence $\beta x = \beta y_\gamma \gamma z \in \delta_{\alpha\beta}(A) \cdot B$. Thus

$$\delta_{\alpha\beta}(AB) = \delta_{\alpha\beta}(A) \cdot B \ \cup \ \bigcup_\gamma \varepsilon_{\alpha\gamma}(A) \cdot \delta_{\gamma\beta}(B).$$

Abstracting over indices, we obtain the matrix equation (ii).

(iii) From (i) and (ii):

$$\begin{aligned}\delta(A^*) &= \delta(1 + AA^*) = 0 + \delta(AA^*) \\ &= \delta(A) \cdot I(A^*) + \varepsilon(A) \cdot \delta(A^*).\end{aligned}$$

The derivative is the least fixpoint of this equation, which by an axiom of KAT is the right-hand side of (iii). □

The following lemma says that G is a coalgebra morphism from the syntactic coalgebra (Exp, D, E) to $(2^U, \delta, \varepsilon)$.

Lemma 4.

(i) $G(D(e)) = \delta(G(e))$
(ii) $E(e) = \varepsilon(G(e))$

where G is extended componentwise to matrices.

Proof. By induction on e.

(i) For primitive terms p, b and dup,

$$\begin{aligned}G(D_{\alpha\beta}(p)) &= G(0) = \varnothing \\ &= \{\beta x \mid \alpha p_\beta \ \mathsf{dup}\ x \in \{\gamma p \mid \gamma \in \mathsf{At}\}\} \\ &= \delta_{\alpha\beta}(\{\gamma p \mid \gamma \in \mathsf{At}\}) = \delta_{\alpha\beta}(G(p)) \\ G(D_{\alpha\beta}(b)) &= G(0) = \varnothing \\ &= \{\beta x \mid \alpha p_\beta \ \mathsf{dup}\ x \in \{\beta p_\beta \mid \beta \le b\}\} \\ &= \delta_{\alpha\beta}(\{\beta p_\beta \mid \beta \le b\}) = \delta_{\alpha\beta}(G(b)). \\ G(D_{\alpha\beta}(\mathsf{dup})) &= G(\alpha \cdot [\alpha = \beta]) \\ &= \{\beta p_\beta \mid \alpha = \beta\} \\ &= \{\beta x \mid \alpha p_\beta \ \mathsf{dup}\ x \in \{\gamma p_\gamma \ \mathsf{dup}\ p_\gamma \mid \gamma \in \mathsf{At}\}\} \\ &= \delta_{\alpha\beta}(\{\gamma p_\gamma \ \mathsf{dup}\ p_\gamma \mid \gamma \in \mathsf{At}\}) \\ &= \delta_{\alpha\beta}(G(\mathsf{dup}))\end{aligned}$$

The case $e_1 + e_2$ is straightforward, since G, δ, and D are linear. For products, using Lemma 3 (ii),

$$\begin{aligned}G(D(e_1e_2)) &= G(D(e_1) \cdot I(e_2)) + G(E(e_1) \cdot D(e_2)) \\ &= G(D(e_1)) \cdot G(I(e_2)) + G(E(e_1)) \cdot G(D(e_2)) \\ &= \delta(G(e_1)) \cdot I(G(e_2)) + \varepsilon(G(e_1)) \cdot \delta(G(e_2)) \\ &= \delta(G(e_1) \cdot G(e_2)) \\ &= \delta(G(e_1e_2))\end{aligned}$$

$$\begin{aligned}\mathit{lrsp}(e_1 + e_2) &= \mathit{lrsp}(e_1) \cup \mathit{lrsp}(e_2) \\ \mathit{lrsp}(e_1e_2) &= \{(\ell, re_2) \mid (\ell, r) \in \mathit{lrsp}(e_1)\} \cup \\ &\qquad \{(e_1\ell, r) \mid (\ell, r) \in \mathit{lrsp}(e_2)\} \\ \mathit{lrsp}(e^*) &= \{(e^*\ell, re^*) \mid (\ell, r) \in \mathit{lrsp}(e)\} \\ \mathit{lrsp}(\mathsf{dup}) &= \{(1, 1)\} \\ \mathit{lrsp}(b) &= \mathit{lrsp}(p) = \varnothing.\end{aligned}$$

Figure 1. NetKAT left-right spines.

For star, the system defining $D(e^*)$ is

$$D(e^*) = D(e) \cdot I(e^*) + E(e) \cdot D(e^*)$$

whose least solution is

$$D(e^*) = E(e^*) \cdot D(e) \cdot I(e^*).$$

Using Lemma 3 (iii),

$$\begin{aligned}G(D(e^*)) &= G(E(e^*) \cdot D(e) \cdot I(e^*)) \\ &= G(E(e^*)) \cdot G(D(e)) \cdot G(I(e^*)) \\ &= \varepsilon(G(e)^*) \cdot \delta(G(e)) \cdot I(G(e)^*) \\ &= \delta(G(e^*)).\end{aligned}$$

(ii) For p, b and dup,

$$\begin{aligned}E_{\alpha\beta}(p) &= [p = p_\beta] \\ &= \varepsilon_{\alpha\beta}(\{\gamma p \mid \gamma \in \mathsf{At}\}) = \varepsilon(G(p)). \\ E_{\alpha\beta}(b) &= [\alpha = \beta \le b] \\ &= \varepsilon_{\alpha\beta}(\{\alpha p_\alpha \mid \alpha \le b\}) \\ &= \varepsilon_{\alpha\beta}(G(b)). \\ E_{\alpha\beta}(\mathsf{dup}) &= 0 \\ &= \varepsilon_{\alpha\beta}(\{\gamma p_\gamma \ \mathsf{dup}\ p_\gamma \mid \gamma \in \mathsf{At}\}) \\ &= E_{\alpha\beta}(G(\mathsf{dup}))\end{aligned}$$

The case $e_1 + e_2$ is straightforward, since G, ε, and E are linear. For products, using Lemma 2 (iii),

$$\begin{aligned}E_{\alpha\beta}(e_1e_2) &= \sum_\gamma E_{\alpha\gamma}(e_1) \cdot E_{\gamma\beta}(e_2) \\ &= (E(e_1) \cdot E(e_2))_{\alpha\beta} \\ &= (\varepsilon(G(e_1)) \cdot \varepsilon(G(e_2)))_{\alpha\beta} \\ &= (\varepsilon(G(e_1) \cdot G(e_2)))_{\alpha\beta} \\ &= \varepsilon_{\alpha\beta}(G(e_1e_2))\end{aligned}$$

For star, using Lemma 2 (iv),

$$E(e^*) = E(e)^* = \varepsilon(G(e))^* = \varepsilon(G(e^*)). \quad \square$$

5.2 Spines

As was just shown, matrices provide an elegant and compact way to express and encode NetKAT derivatives. It turns out that the set of derivatives of an expression is finite and can be bounded as a function of the size of the expression itself. To prove this, we develop the notion of the *spines* of a term e and show that derivatives can always be represented as sums of spines. In our implementation, both of these representations are put to work: we encode the Brzozowski derivative using matrices whose elements are sets of spines.

Intuitively, the spines of an expression can be obtained by locating the occurrences of dup and forming a pair of expressions built from the expressions appearing to the left and right of the

dup. The left component of the pair is called the *left spine* and the right component is called the *right spine*. The spines are related to the derivative in the following way: the left spine represents the expression that must be consumed before the occurrence of dup can be consumed itself, and the right spine indicates the expression that remains after doing so. For example, the set of spines of the expression $a \cdot \mathsf{dup} \cdot b$ is just $\{(a, b)\}$, and indeed, a is the expression that must be consumed before the dup and b is the expression that remains after it is consumed.

The inductive definition of the left-right spines of e, denoted $lrsp(e)$, is given in Figure 1. In many situations, just the right spines are useful. They can be defined more simply as follows (to lighten the notation, we write $A \cdot e$ for $\{de \mid d \in A\}$ and $e \cdot A$ for $\{ed \mid d \in A\}$ where $A \subseteq \mathsf{Exp}$ and $e \in \mathsf{Exp}$):

$$\begin{aligned} rsp(e_1 + e_2) &= rsp(e_1) \cup rsp(e_2) \\ rsp(e_1 e_2) &= rsp(e_1) \cdot e_2 \cup rsp(e_2) \\ rsp(e^*) &= rsp(e) \cdot e^* \\ rsp(\mathsf{dup}) &= \{1\} \\ rsp(b) = rsp(p) &= \varnothing. \end{aligned}$$

It is easy to show that every right spine in $rsp(e)$ has the form $1 \cdot e_1 \cdot e_2 \cdots e_n$, where the e_i are subexpressions of e, and that there is one spine of e for every occurrence of dup in e.

The next lemma relates the derivative of e and its right spines:

Lemma 5. *For any α, β,*

$$D_{\alpha\beta}(e) = \{\beta r \mid (\ell, r) \in lrsp(e),\ E_{\alpha\beta}(\ell) = 1\}.$$

Thus the derivative $D_{\alpha\beta}(e)$ is a sum of terms of the form βr, where $r \in rsp(e)$.

Proof. The proof is by induction on the structure of e. Abusing notation slightly by representing sums of terms as sets,[4] we argue the cases for products and star explicitly.

For products, we have the following equalities:

$$\begin{aligned} &D_{\alpha\beta}(e_1 e_2) \\ &= D_{\alpha\beta}(e_1) \cdot e_2 \cup \textstyle\bigcup_{E_{\alpha\gamma}(e_1)=1} D_{\gamma\beta}(e_2) \\ &= \{\beta r \mid (\ell, r) \in lrsp(e_1),\ E_{\alpha\beta}(\ell) = 1\} \cdot e_2 \\ &\quad \cup \textstyle\bigcup_{E_{\alpha\gamma}(e_1)=1} \{\beta r \mid (\ell, r) \in lrsp(e_2),\ E_{\gamma\beta}(\ell) = 1\} \\ &= \{\beta r e_2 \mid (\ell, r) \in lrsp(e_1),\ E_{\alpha\beta}(\ell) = 1\} \\ &\quad \cup \{\beta r \mid (\ell, r) \in lrsp(e_2),\ E_{\alpha\beta}(e_1 \ell) = 1\} \\ &= \{\beta r \mid (\ell, r) \in lrsp(e_1 e_2),\ E_{\alpha\beta}(\ell) = 1\}, \end{aligned}$$

where we use the induction hypothesis in the second step.

For star, we have the following equalities:

$$\begin{aligned} &D_{\alpha\beta}(e^*) \\ &= \textstyle\bigcup_{E_{\alpha\gamma}(e^*)=1} D_{\gamma\beta}(e) \cdot e^* \\ &= \textstyle\bigcup_{E_{\alpha\gamma}(e^*)=1} \{\beta r \mid (\ell, r) \in lrsp(e),\ E_{\gamma\beta}(\ell) = 1\} \cdot e^* \\ &= \{\beta r e^* \mid (\ell, r) \in lrsp(e),\ E_{\alpha\beta}(e^* \ell) = 1\} \\ &= \{\beta r \mid (\ell, r) \in lrsp(e^*),\ E_{\alpha\beta}(\ell) = 1\}. \end{aligned}$$

□

The final lemma presented in this section shows that the spines of spines of e are themselves spines of e. Hence, taking repeated derivatives does not introduce new terms.

Lemma 6. *If $d \in rsp(e)$, then $rsp(\beta d) \subseteq rsp(e)$.*

Proof. The argument for sums is straightforward. For products,

$$\begin{aligned} &d \in rsp(e_1 e_2) = rsp(e_1) \cdot e_2 \cup rsp(e_2) \\ &\Rightarrow d \in rsp(e_1) \cdot e_2 \text{ or } d \in rsp(e_2) \\ &\Rightarrow (d = ce_2 \text{ and } c \in rsp(e_1)) \text{ or } d \in rsp(e_2) \\ &\Rightarrow (d = ce_2 \text{ and } rsp(\beta c) \subseteq rsp(e_1)) \\ &\quad \text{or } rsp(\beta d) \subseteq rsp(e_2) \\ &\Rightarrow rsp(\beta d) = rsp(\beta c e_2) = rsp(\beta c) \cdot e_2 \cup rsp(e_2) \\ &\quad \subseteq rsp(e_1) \cdot e_2 \cup rsp(e_2) = rsp(e_1 e_2) \\ &\quad \text{or } rsp(\beta d) \subseteq rsp(e_2) \subseteq rsp(e_1 e_2) \\ &\Rightarrow rsp(\beta d) \subseteq rsp(e_1 e_2). \end{aligned}$$

For star,

$$\begin{aligned} &d \in rsp(e^*) = rsp(e) \cdot e^* \\ &\Rightarrow d = ce^* \text{ and } c \in rsp(e) \\ &\Rightarrow rsp(\beta d) = rsp(\beta c e^*) = rsp(\beta c) e^* \cup rsp(e^*) \\ &\quad \subseteq rsp(e) \cdot e^* \cup rsp(e^*) = rsp(e^*). \end{aligned}$$

For dup,

$$\begin{aligned} &d \in rsp(\mathsf{dup}) = \{1\} \\ &\Rightarrow d = 1 \\ &\Rightarrow rsp(\beta d) = rsp(\beta) = \varnothing \subseteq rsp(\mathsf{dup}). \end{aligned}$$

Note that we cannot have $d \in rsp(b)$ or $d \in rsp(p)$, since these sets are empty. □

Taken together, these lemmas show that repeated derivatives of e can all be represented as sums of terms of the form βd, where $d \in rsp(e)$. Thus the number of derivatives of e is at most $|\mathsf{At}| \cdot 2^\ell$, where ℓ is the number of occurrences of dup in e. Moreover, these terms can be represented compactly as a pair of an atom and a subset of $rsp(e)$. Using these representations to build NetKAT automata provides a solid foundation for building an efficient implementation, as is described in the next section.

6. Implementation

We have built a system that decides NetKAT equivalence. Given two NetKAT terms, it first converts these terms into automata using Brzozowski derivatives, and then tests whether the automata are bisimilar. Our implementation consists of 4500 lines of OCaml code and includes a parser, pretty printer, and a simple visualizer. We have also integrated our decision procedure into the Frenetic SDN controller platform. This integration enables automated verification of important properties for real-world network topologies and configurations.

Our implementation incorporates a number of important enhancements and optimizations that avoid potential sources of combinatorial blowup. In particular, the derivative and matrix-based algorithms described in the preceding sections are formulated in terms of the NetKAT language model consisting of sets of reduced strings of complete tests and assignments. Building a direct implementation of these algorithms would require constructing square matrices indexed by the universe of possible complete tests and assignments, which is exponential in the number of constants in the terms. Following such a strategy would be impractical, even for small terms. Instead, our implementation uses a symbolic representation that exploits symmetry and sparseness and incorporates optimizations to aggressively prune away values that do not contribute to the final outcome. Although the algorithm is still exponential in the worst case—which is unavoidable, as the problem is PSPACE-complete—the constrained nature of real-world networks allows our tool to perform well in many common cases.

[4] This is a convenient abuse which we take with impunity as we are working modulo ACI. The representation of the Brzozowski derivative in this form is often called the *Antimirov derivative*.

6.1 Data Structures

The foundation of our implementation is based on a collection of data structures that provide symbolic representations for building and analyzing NetKAT automata.

Bases. Bases represent sets of pairs of complete tests and assignments symbolically, typically avoiding having to enumerate every possible packet value. Let e be a NetKAT term and let $x_1, \ldots, x_n$ be the collection of fields appearing in it. Likewise, let U_i be the universe of all values associated with x_i either by a test $x_i = n$ or an assignment $x_i \leftarrow n$. A *base* is a pair of sequences $X_1, \ldots, X_m; o_1, \ldots, o_m$, where the $X_i \subseteq U_i$ are sets of values and the $o_i \in U_i$ are optional values. The set represented by a base contains all tests where the value of the test for field x_i is drawn from X_i and the value for the assignment to x_i is either n_i if o_i is defined and equal to n_i, or the same value as the test otherwise.

Matrices. Using bases, it is straightforward to build a sparse matrix representation in which the rows and columns are indexed by complete tests and assignments. To encode a 0-1 matrix, we simply use a set of bases. To encode a matrix over a set, we use finite maps from bases to elements of the set. For example, when constructing the E matrix for a term e, tests $x_i = m_i$ are represented by

$$U_1, \ldots, U_{i-1}, \{m_i\}, U_{i+1}, \ldots, U_n; ?, \ldots, ?,$$

where ? denotes a missing optional value. Similarly, assignments $x_i \leftarrow m_i$ are represented by

$$U_1, \ldots, U_n; ?, \ldots, ?, m_i, ?, \ldots, ?.$$

Sums and products can be obtained using matrix addition and multiplication as implemented using base sets. The product of bases $(X_1, \ldots, X_n; o_1, \ldots, o_n)$ and $(Y_1, \ldots, Y_n; q_1, \ldots, q_n)$ is nonzero if there exists a complete assignment in the left base that matches a complete test in the right for each field. If $o_i = ?$, then the intersection of X_i and Y_i must be nonempty, otherwise the tests corresponding to the ith field will drop all packets produced by the left base. On the other hand, if $o_i \neq ?$, then its value must belong to Y_i. If these conditions hold, the resulting product $Z_1, \ldots, Z_n; w_1, \ldots, w_n$ is defined as follows:

$$Z_i = \begin{cases} X_i & \text{if } o_i \neq ? \\ X_i \cap Y_i & \text{if } o_i = ? \end{cases} \qquad w_i = \begin{cases} o_i & \text{if } o_i \neq ? \text{ and } q_i = ? \\ q_i & \text{if } q_i \neq ? \\ ? & \text{if } o_i = q_i = ? \end{cases}$$

Using the product operation on bases, it is easy to build other matrix operations. For example, multiplication can be implemented by folding over the base sets, and fixpoints can be computed using an iterative loop that multiplies at each step or by repeated squaring.

6.2 Algorithms

The two core pieces of our implementation are (i) an algorithm that computes automata using Brzozowski derivatives, and (ii) another that checks bisimilarity of automata.

Brzozowski derivative. Our implementation of Brzozowski derivatives uses the *spines* introduced in §5.2. Recall that there is one spine for every occurrence of dup in e. If σ denotes an occurrence of dup, let ℓ_σ and r_σ denote the left spine and right spine, respectively, of that occurrence. It is straightforward to show that

$$D_{\alpha\beta}(e) = \bigcup\{\beta r_\sigma \mid \sigma \text{ an occurrence of } \mathsf{dup},\ E_{\alpha\beta}(\ell_\sigma) = 1\}$$

or more succinctly,

$$D(e) = \sum_\sigma E(\ell_\sigma) \cdot J \cdot I(r_\sigma). \tag{6.1}$$

To further streamline the computation of $D(e)$, we can avoid adding the βr_σ term to $D_{\alpha\beta}(e)$ when βr_σ is zero, or equivalently when there exists no element of $G(r_\sigma)$ of the form βx. Let Φ be a function that replaces all occurrences of dup with 1 as follows:

$$\Phi(p) = p \qquad \Phi(b) = b \qquad \Phi(\mathsf{dup}) = 1$$

$$\Phi(e_1 + e_2) = \Phi(e_1) + \Phi(e_2)$$

$$\Phi(e_1 e_2) = \Phi(e_1) \cdot \Phi(e_2) \qquad \Phi(e^*) = \Phi(e)^*.$$

It is easy to show that for any α, the set $\{\alpha \mid \alpha x \in G(e)\}$ is equal to $\{\alpha \mid \alpha x \in G(\Phi(e))\}$. Hence, $\beta x \notin G(r_\sigma)$ is equivalent to $\beta x \notin G(\Phi(r_\sigma))$. Moreover, because $\Phi(r_\sigma)$ does not contain dup, the set $\{\alpha \mid \alpha x \in G(e)\}$ is described by the left-hand sequences $(A_1, \ldots, A_n)$ in the base set representation of $E(\Phi(r_\sigma))$. Hence the derivative can be described as:

$$D(e) = \sum_\sigma E(\ell_\sigma) \cdot E'(\Phi(r_\sigma)) \cdot J \cdot I(r_\sigma),$$

where $E'(\Phi(r_\sigma))$ is obtained from $E(\Phi(r_\sigma))$ by replacing each base $(A_1, \ldots, A_n; k_1, \ldots, k_n)$ with $(A_1, \ldots, A_n; ?, \ldots, ?)$.

This formulation, which is used in our implementation, has a number of advantages. First, it is expressed entirely in terms of simple matrix operations involving the E, I, and J matrices. Second, it aggressively filters away intermediate terms that do not contribute to the overall result. In particular, if the $\alpha\beta$ entry of $E(\ell_\sigma)$ is 0, then the occurrence of dup indicated by σ does not contribute to the first dup in any reduced string denoted by e, so the derivative is also 0. Third, since the spines of spines of e are spines of e, we can calculate the left-right spines once and for all when we construct the term, and subsequent derivatives are guaranteed to have the form of (6.1).

Bisimulation. The other step in our NetKAT decision procedure tests the bisimilarity of the automata constructed using Brzozowski derivatives. Once we have the coalgebraic structure, this algorithm is fairly standard. Given two NetKAT terms e_1 and e_2, we first compare the matrices $E(e_1)$ and $E(e_2)$ and check whether they are identical, returning false immediately if they are not. Otherwise, we calculate all derivatives of e_1 and e_2 and recursively check each of the resulting pairs. The algorithm halts when we have tested every possible derivative reachable transitively from the initial terms. The bounds derived in §5.2 guarantee that the algorithm terminates. This coinductive algorithm can be implemented in almost linear time in the combined size of the automata using the union-find data structure [13].

6.3 Optimizations

To further improve performance, our implementation incorporates a number of optimizations designed to reduce the overhead of representing and computing with terms, bases, sparse matrices, etc.

Hash consing and memoization. Encoding real-world network topologies and configurations as NetKAT terms often leads to many repetitions. Our implementation exploits this insight by using aggressive normalization and hash consing so that (many) semantically equivalent terms are represented by the same syntactic term. For instance, we represent products as lists, which gives multiplicative associativity for free, and we represent sums as sets, which gives additive associativity, commutativity, and idempotence for free. We also use smart constructors that recognize identities involving 0 and 1 (along with several others) to further optimize the representations of terms. We calculate term metadata such as left-right spines and the E matrix lazily and store the results alongside the term itself.

Sparse multiplication. The straightforward way to represent matrix multiplication in terms of base sets would be to use nested folds over the sets—the product of two base sets is simply their

point-wise cross-product. Sadly, this naive algorithm gives poor performance, and much of the effort is wasted multiplying bases whose product is always 0. We instead implement an algorithm which, rather than iterating over all pairs of bases, instead proactively filters the sets, only retaining elements which could produce a nonzero result. This is done as follows. (i) We first build an association from assignments to their originating bases from the left matrix. (ii) For each base in the right matrix, we intersect all test value sets in this base with the set of all assignments in the left matrix (per field). We will call the result of this operation the set of potential matches. (iii) We multiply each base in the right matrix with the bases corresponding to its potential matches to produce the overall product. These operations allow us to associate each base in the right operand with a small set of left operand bases, significantly limiting the number of pairs we multiply.

Base compaction. There can be many representations of a set of complete tests and assignments, and as base sets are multiplied, those representations tend to grow. Hence, another key optimization for making matrix multiplication (and many other operations) fast is to compact the base sets whenever possible. Two bases can be merged when (i) one is a subset of the other or (ii) they are adjacent, in the following senses. The base $b_1 = X_1, \ldots, X_n; o_1, \ldots, o_n$ is a *subset* of base $b_2 = Y_1, \ldots, Y_n; q_1, \ldots, q_n$ when for all fields i, we have $X_i \subseteq Y_i$ and $o_i = q_i$. In this case, b_1 and b_2 can be replaced with just b_2. The base b_1 is *adjacent* to b_2 if there exists a field i such that $o_i = q_i$ and for all other fields j both $X_j = Y_j$ and $o_j = q_j$. The result of merging these two bases is $Y_1, \ldots, X_i \cup Y_i, \ldots, Y_n; q_1, \ldots, q_n$. Although both of these merging optimizations require bases with identical assignments, we can efficiently reduce the number of bases we attempt to merge by sorting base sets by their assignments. This yields a fast optimization that dramatically compacts the base sets that must be maintained in our implementation.

Fast fixpoints. The algorithm for calculating the E matrix presented in §5.1 uses a fixpoint, which is a potentially expensive operation. Our implementation incorporates several optimizations that greatly increase the efficiency of calculating this fixpoint by exploiting the structure of terms encoding networks. Generally speaking, network terms are of the form $in \cdot (p \cdot t)^* \cdot p \cdot out$, where in and out are edge policies that describe the packets entering and leaving the network, p is a policy the describes the behavior of the switches, and t encodes the topology. The edge policies are typically very small compared to p or t. Hence, we can use the edge policies to cut down the size of the $(p \cdot t)^*$ term as we take its fixpoint. We first unfold $in \cdot (p \cdot t)^* \cdot p \cdot out$ to $in \cdot out + in \cdot (p \cdot t)^* \cdot out$ and then find the $in \cdot (p \cdot t)^*$ fixpoint by calculating $in \cdot p \cdot t + in \cdot p \cdot t \cdot p \cdot t + \cdots + in \cdot (p \cdot t)^i$, stopping when we have reached a fixpoint. We have determined empirically that when in and out are relatively small this process converges much faster than other techniques for computing fixpoints, such as repeatedly squaring the $(p \cdot t)$ term.

7. Additional Applications

The utility of NetKAT's coalgebraic theory is not limited to checking equivalence via bisimulation. The E and D matrices are also useful for solving many other practical verification problems directly. This section discusses several such applications: all-pairs connectivity, loop freedom, and translation validation.

Connectivity. Reachability—whether host A can communicate with host B—is a fundamental property of a network. Indeed, there are now many automated tools that can check reachability properties involving individual locations; see §9 for a survey. Connectivity is a slightly stronger property that asks whether *every* host in a network can communicate with every other host. In other words, connectivity tests whether a network provides the functionality of one "big switch" that forwards traffic between all of its ports.

As connectivity cares only about the end-to-end properties of a network, it is agnostic to paths. Hence, it can be modeled in NetKAT by simply setting all occurrences of dup to 1 using Φ from §6 and checking the following:

$$\Phi(in \cdot (p \cdot t)^* \cdot p \cdot out) = \sum_{(sw, sw', pt, pt')} \begin{pmatrix} switch = sw \cdot port = pt \cdot \\ switch \leftarrow sw' \cdot port \leftarrow pt' \end{pmatrix}$$

where p encodes the switch policy, t encodes the topology, and in the summation, (sw, pt) and (sw', pt') range over all outward-facing switch-port pairs, that is, those adjacent to a host. Intuitively, the left-hand side of the equation encodes the end-to-end forwarding behavior of the network—forwarding that starts from a state matching in and traverses the switch policy and topology any number of times and eventually reaches a state matching out—while the right-hand side represents a specification of a network that forwards directly between every outward-facing switch-port pair. Because connectivity does not involve keeping track of paths, it can be verified simply by checking equality of E matrices.

Forwarding loops. A network has a *forwarding loop* if some packets traverse a cycle repeatedly. Forwarding loops are a frequent source of error in networks and have been identified as the cause of outages both in local-area networks, where loops are often produced by protocols for computing broadcast spanning trees, and on the broader Internet, where inter-domain protocols such as BGP can easily produce loops during periods of reconvergence [15]. Making matters worse, forwarding loops are often masked by features such as TTL (time-to-live) fields, a run-time mechanism that enforces an upper bound on the length of any loop by decrementing a counter at each hop and dropping the packet when the counter reaches 0.

To check whether a packet has a loop, we need to determine if there exists a packet that can reach the same location with the same value twice. One possible way to express this is with the equation $in \cdot (p \cdot t)^{2^n} = 0$, where p is the switch policy, t is the topology, and n is the number of complete tests which occur in the program. The intuition behind this equation is that there are only 2^n distinct packet values, so any packet that traverses the network more than 2^n times must have been in some state at least twice, and thus will loop forever. However, while this equation is correct, checking it directly is inefficient, as exponentiation is an expensive operation.

To make the problem more tractable, we can recast it into an equivalent formulation using a quantifier:

$$\forall \alpha.\ prefix(in \cdot (p \cdot t)^*) \cdot \alpha \cdot p \cdot t \cdot (p \cdot t)^* \cdot \alpha = 0,$$

where *prefix* is the prefix closure operation. This equation directly encodes a packet that visits the same state twice, allowing us to avoid an expensive exponentiation operation. Moreover, it is not hard to show that the inner term is already prefix-closed, as it ends with a star. So we can reformulate our test to:

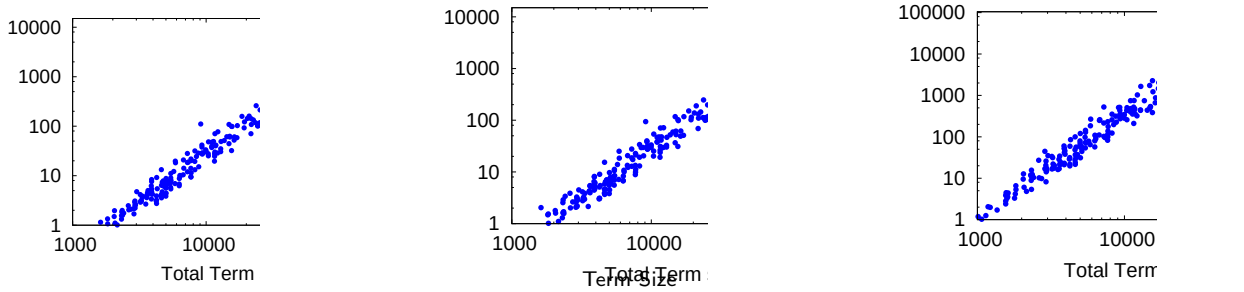
$$\forall \alpha.\ in \cdot (p \cdot t)^* \cdot \alpha \cdot p \cdot t \cdot (p \cdot t)^* \cdot \alpha = 0.$$

After converting to matrices, we obtain:

$$\forall (\alpha, \beta) \in E(\Phi(in \cdot (p \cdot t)^*)).\ \beta \cdot (p \cdot t) \cdot (p \cdot t)^* \cdot \beta = 0.$$

We then observe that for any complete test β and term x, we have $\beta \cdot x \cdot \beta = 0$ if and only if $E_{\beta\beta}(\Phi(x)) = 0$. This yields a fast algorithm for determining loop freedom. We iterate through the sets of possible α and β with nonzero entries in $E(\Phi(in \cdot (p \cdot t)^*))$ and check the entry in $E(\Phi(p \cdot t \cdot (p \cdot t)^*))$. If it is also nonzero, then the network has a loop. We have used this algorithm to check for loops

Topology	Term Size	# Switches	Largest Policy	Parse Time	Loop Freedom	Connectivity	Translation
Airtel	2412	15	112	0.102	3.386	1.755	10.424
Uran	5000	23	264	0.327	5.905	5.757	71.18
GtsSlovakia	10388	34	385	0.99	25.361	25.166	258.83
Uunet	20350	48	539	6.684	138.147	143.387	2007.68
Telcove	41474	72	781	32.826	280.413	300.265	6959.08
Oteglobe	56844	92	838	93.045	944.955	944.555	26292.8
Pern	131092	126	4757	311.644	2140.63	2478.24	83245.7



(a) connectivity (b) loop freedom (c) translation validation

Figure 2. Topology Zoo experimental results.

in networks with topologies containing thousands of switches and configurations with thousands of forwarding rules on each switch.

Translation validation. One technique for checking the correctness of a compiler, often called *translation validation*, is to test whether the instructions it emits have same semantics as the programs provided as input [29]. We can use translation validation to check the NetKAT compiler itself, which is used by the Frenetic controller [11]. Unlike the applications just discussed, which only depend on analyzing E matrices and do not require bisimulation, translation validation uses the full NetKAT decision procedure. This is due to the fact that it must check the equivalence of the paths generated by the compiler rather than just checking end-to-end forwarding.

We have developed a simple application that uses bisimulation to validate the output of the Frenetic compiler. It takes an input policy p and invokes the NetKAT compiler to convert it to a sequence of OpenFlow forwarding rules, one for each switch. As was shown in the original NetKAT paper [1], the language is expressive enough to encode these rules, so we can reflect them back into NetKAT terms as nested cascades of conditionals,

$$\begin{aligned} c \quad = \quad & \mathsf{if}\ pat_1\ \mathsf{then}\ acts_1\ \mathsf{else} \\ & \cdots \\ & \mathsf{if}\ pat_k\ \mathsf{then}\ acts_k\ \mathsf{else}\ 0 \end{aligned}$$

where each pat_i is a positive conjunction of tests and each act_i is a sequence of modifications. To verify equivalence, we simply check whether $p = (c \cdot t)^* \cdot c$, where t is the topology. If this succeeds, we know that the forwarding rules emitted by the compiler encode the same paths as those specified in the program.

8. Evaluation

To evaluate the performance of our implementation, we conducted experiments on a variety of benchmarks. These experiments were designed to answer the following questions: Can our coalgebraic decision procedure effectively answer practical questions about real-world network topologies and configurations? How does its performance scale as the inputs grow in size? How does its performance compare to other network verification tools?

Benchmarks. We ran experiments on the following benchmarks:

- *Topology Zoo* [18]: This public dataset contains a collection of 261 actual network topologies from around the world, mostly for regional ISPs and carrier networks. The topologies range in size from 5 nodes (ARPANET) to over 196 nodes (Cogentco).[5] We generated forwarding policies by placing hosts into the topology at random and computing paths that forward between all pairs of hosts. This benchmark provides the ability to experiment with a wide variety of topologies, ranging from trees, to stars, to meshes, and even seemingly random structures.
- *FatTrees*: These topologies, which are commonly used in data center networks, consist of a tree-structured hierarchy in which the switches at each level have multiple redundant connections to switches at the next higher level. We wrote a Python script to generate FatTrees for a given pair of depth and fanout parameters as well as a NetKAT policy that provides connectivity between hosts. This benchmark provides the ability to experiment with the scalability of our tool on a commonly used topology at varying sizes.
- *Stanford Backbone* [16]: This includes the 16-node topology of the Stanford campus backbone network as well as the actual configurations of each router. This benchmark provides an example of a complete real-world network and provides a means to compare performance directly against other verification tools, such as Header Space Analysis (HSA) [16].

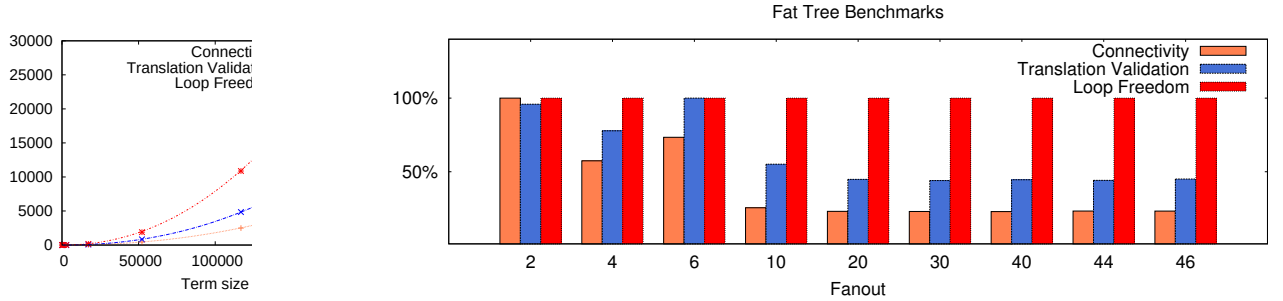
Properties. Our experiments focused on checking the following properties and used the applications described in §7:

- *Connectivity*: Does the network establish connectivity between all pairs of hosts?
- *Loop-Freedom*: Does the network have forwarding loops?
- *Translation Validation*: Does the NetKAT compiler translate high-level policies into equivalent OpenFlow forwarding rules?

For the Stanford benchmark, to facilitate a head-to-head comparison with HSA, we also performed a reachability query from a source to single destination.

[5] We omitted one outlier topology, Kdl, which is the largest topology and significantly larger than the next-largest topology in the Zoo.

Fanout	Depth	Term Size	# of Switches	Largest Policy	Parse Time	Connectivity	Translation	Loop Freedom
4	2	311	6	16	0.004000	0.031	0.042	0.054
10	2	2807	15	590	0.052003	0.699997	1.511	2.742
20	2	17207	30	3880	0.636040	38.246	74.17	165.548
30	2	52207	45	8670	6.256391	435.082	832.18	1891.75
46	2	173519	69	21781	126.051878	5858.85	11338.7	25179.1



(a) scalability (b) relative performance

Figure 3. FatTree experimental results.

Methodology. We ran our experiments on a small cluster of Dell r720 servers with four eight-core 2.70GHz Intel Xeon CPU E5-2680 processors and 64GB of RAM running Ubuntu 12.04.4 LTS. We restricted each experiment to run on a single core. We collected running times using the Unix `time` command for total process times and the OCaml function `Sys.time()` for sub-process times. In each experiment, we exclude the time required to parse inputs and generate policies and only report the amount of time used for the actual verification task.

Results and Analysis. The results of our experiments on the Topology Zoo and FatTree benchmarks are depicted in Figures 2 and 3. We plot the running times of the benchmarks on inputs of varying size and also provide a sampling of data points in a table. All times are reported in seconds.

For the Topology Zoo benchmark, we see that our implementation is able to check small topologies with dozens of switches and policies with hundreds of rules on each switch in tens of seconds, and it scales to topologies with hundreds of switches and policies with thousands of rules without difficulty. The graphs in Figure 2 plot the running time of all three applications against the size of the NetKAT input term. Note that the plots use a semi-logarithmic scale to show the overall trend. The performance of loop detection is similar to connectivity. The performance of translation validation is slower, taking about an order of magnitude longer on large inputs, taking tens of hours for topologies with thousands of switches. This is expected, as translation validation involves invoking the full bisimulation algorithm, a step not required to perform most practical verification tasks. However, our tool is still able to complete and produce the correct result.

For the FatTree benchmark, we measured the scalability of our tool as the size of the network increases from a small number of nodes to several hundred nodes. The graph on the left of Figure 3 plots the performance of all three applications against the size of the NetKAT input term. We observe similar scaling as for the Topology Zoo benchmarks: small networks complete in seconds, while larger networks can take up to several hours. The graph on the right of Figure 3 compares the performance of our three applications, giving the relative running time compared to the slowest application—loop freedom—on FatTrees of increasing size. On small inputs, all three applications take roughly the same amount of time, whereas on larger inputs, connectivity is fastest, loop detection is about half as fast, and translation validation is again half as fast.

For our final benchmark, we compared the running time of our tool against the HSA network property-checking tool [16]. HSA works by doing a symbolic ternary simulation of the space of possible packet values through the policy. It incorporates numerous optimizations to prune the space and keep the representation compact. HSA is able to answer simple queries like reachability and loop detection involving a single host or port, but it does not check full equivalence, unless one performs iterated reachability queries over the entire state space. Nevertheless, for many properties of interest, HSA is able to produce an answer in a few seconds.

To facilitate a comparison with HSA, we made several further improvements to our tool. First, we wrote a front end to parse router configurations for the Stanford backbone. Second, we wrote a tool to convert configurations based on IP prefix matching into policies that only test concrete IP values. This tool works by computing a partitioning of the space of all possible IP addresses that respects the constants mentioned in the program and then replacing each IP prefix with the union of representatives of the equivalence classes it includes. Third, we developed an optimization that statically analyzes NetKAT policies and determines which fields are static, using the NetKAT axioms and partial evaluation to considerably reduce the size of the search space. For example, if a policy matches on IP destination addresses and never modifies those addresses, then for any particular address we can partially evaluate the policy to obtain a smaller policy that is specialized to that host. This analysis and optimization is integrated into our algorithm for constructing NetKAT automata and applied automatically during the computation of fixpoints for Kleene star.

With these enhancements, our tool is able to answer a reachability query involving a single source host in 0.67 seconds. Note that this query is evaluated on the production Stanford backbone network with 16 routers and thousands of forwarding rules. In the original HSA paper, the authors report 13 seconds for a single reachability query. We were able to replicate this number on our own testbed. Since publishing their original paper, the authors have built a hand-optimized version of HSA in C that can answer the same query in 0.5 seconds, but we were not able to replicate this figure at the time of the writing of this paper.

Discussion. Overall, our experiments demonstrate that our tool for deciding NetKAT equivalence is able to scale to real-world network topologies and configurations and provides good performance on many common properties. Although the general problem

of NetKAT equivalence is PSPACE complete, our implementation is still fast enough to be used for offline verification of production networks and can answer simpler questions such as point-to-point reachability in well under a second, making it also suitable for more dynamic situations.

9. Related Work

NetKAT [1] is the latest in a series of domain-specific languages for SDN programming developed as a part of the Frenetic project [11, 12, 26, 27]. NetKAT largely inherits its syntax, semantics, and application methodology from these earlier efforts but adds a complete deductive system and PSPACE decision procedure [1]. These new results in NetKAT build on the strong connection to earlier algebraic work in KA and KAT [21, 22, 24]. The present paper extends work on NetKAT further, developing the coalgebraic theory of the language and engineering an implementation of these ideas in an OCaml prototype. The overall result is the first practical implementation for deciding NetKAT equivalence.

The coalgebraic theories of KA and KAT and related systems have been studied extensively in recent years [8, 23, 33, 36], uncovering strong relationships between the algebraic/logical view of systems and the combinatorial/automata-theoretic view. We have exploited these ideas heavily in the development of NetKAT coalgebra and NetKAT automata. Finally, in our implementation, we have borrowed many ideas and optimizations from the coalgebraic implementations of KA and KAT and other related systems [4, 5, 30] to provide enhanced performance, making automated decision feasible even in the face of PSPACE completeness. Recent work by Pous developed symbolic techniques for constructing KAT automata and deciding equivalence using binary-decision diagrams (BDDs) [31]. It would be interesting to investigate extending these techniques to NetKAT in the future.

A large number of languages for SDN programming have been proposed in recent years. Nettle [39] applies ideas from functional reactive programming to SDN programming, and focuses on making it easy to express dynamic programs using time-varying signals rather than event loops and callbacks as in most other systems. PANE [10] exposes an interface that allows individual hosts in a network to request explicit functionality such as increased bandwidth for a large bulk transfer or bounded latency for a phone call. Internally PANE uses hierarchical tables to represent and manage the set of requests and a compiler inspired by NetCore [26]. Maple [40] provides a high-level programming interface that enables programmers to express network programs directly in Java, using a special library to match and modify packet headers. Under the hood, the Maple compiler builds up representations of network traffic flows using a tree structure and then compiles these to hardware-level forwarding rules. Several different network programming languages based on logic programming have been proposed including NDLog [25] and FlowLog [28]. The key difference between all of these languages and the system presented in this paper is that NetKAT has a formal mathematical semantics along with a sound and complete deductive apparatus that supports automated reasoning about program equivalence.

Lastly, there is a growing body of work focused on applications of formal methods ranging from lightweight testing to full-blown verification to SDN. The NICE tool [7] uses a model checker and symbolic execution to find bugs in network programs written in Python. Automatic Test Packet Generation [42] constructs a set of packets that provide coverage for a given network-wide configuration. The SDN Troubleshooting System [35] uses techniques inspired by delta debugging to reduce bugs to minimal causal sequences. The VeriCon [2] system uses first-order logic and a notion of admissible topologies to automatically check network-wide properties. It uses the Z3 SMT solver as a back-end decision procedure. Several different systems have proposed techniques for checking network reachability properties including seminal work by Xie et al. [41], Header Space Analysis [16], and VeriFlow[17]. These tools either translate reachability problems into problem instances for other tools, or they use custom decision procedures that extend basic satisfiability checking or ternary simulation with domain-specific optimizations to obtain improved performance. Compared to these tools, NetKAT is unique in its focus on algebraic and coalgebraic structure of network programs. Moreover, as shown in the original NetKAT paper, many properties including reachability can be reduced to equivalence.

10. Conclusion

This paper develops the coalgebraic theory of NetKAT and a new decision procedure based on bisimulation. The coalgebraic theory includes a definition of NetKAT automata, a variant of the Brzozowski derivative, and a version of Kleene's theorem relating terms and automata. A novel aspect of the theory is the concise representation of the Brzozowski derivative in terms of matrices and spines. Our implementation improves on a previous naive algorithm [1] and initial experimental results are promising. In the future, we intend to continue to make further enhancements and perform extensive testing on additional practical examples [1]. A straightforward extension is to incorporate well-studied algorithmic enhancements to the bisimulation construction such as up-to techniques [4, 32]. We also plan to explore extending alternative algorithms for deciding equivalence of KAT expressions [6, 37]. Another possible direction is to study nondeterministic NetKAT automata, which could provide more compact representations, or algorithms for deciding equivalence [3, 4]. We also intend to deploy our tool in the Frenetic SDN controller [11].

Acknowledgments. The authors wish to thank our angel, Ralf Hinze, as well as Nikolaj Bjørner, Rebecca Coombes, Arjun Guha, Andrew Myers, Mark Reitblatt, Ross Tate, Konstantinos Mamouras, Dimitrios Vytiniotis, and the Cornell PLDG for many insightful discussions and helpful comments. Our work is supported by the National Security Agency; the National Science Foundation under grants CNS-1111698, CNS-1413972, DGE-1144153, and SHF-1422046; the Office of Naval Research under grant N00014-12-1-0757; the Dutch Research Foundation (NWO) under project numbers 639.021.334 and 612.001.113; a Sloan Research Fellowship; and a gift from Fujitsu Labs.

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 *POPL*, pages 113–126, January 2014.

[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*, pages 282–293, June 2014.

[3] Filippo Bonchi, Marcello M. Bonsangue, Jan J. M. M. Rutten, and Alexandra Silva. Brzozowski's algorithm (co)algebraically. In *Logic and Program Semantics–Essays Dedicated to Dexter Kozen on the Occasion of His 60th Birthday*, pages 12–23, April 2012.

[4] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *POPL*, pages 457–468, January 2013.

[5] Thomas Braibant and Damien Pous. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, 8(1:16):1–42, 2012.

[6] Sabine Broda, António Machiavelo, Nelma Moreira, and Rogério Reis. On the average size of Glushkov and equation automata for KAT expressions. In *FCT*, pages 72–83, August 2013.

[7] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *NSDI*, April 2012.

[8] Hubie Chen and Riccardo Pucella. A coalgebraic approach to Kleene algebra with tests. In *CMCS*, pages 94–109, July 2003.

[9] 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.

[10] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *SIGCOMM*, pages 327–338, August 2013.

[11] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, pages 279–291, September 2011.

[12] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, pages 483–494, 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, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, pages 3–14, August 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*, pages 395–406, August 2012.

[16] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, April 2012.

[17] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, April 2013.

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

[19] 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*, April 2014.

[20] 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*, pages 351–364, October 2010.

[21] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, May 1994.

[22] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[23] 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.

[24] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In *CSL*, pages 244–259, September 1996.

[25] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, pages 289–300, August 2005.

[26] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, pages 217–230, January 2012.

[27] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *NSDI*, April 2013.

[28] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, April 2014.

[29] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, pages 151–166, March 1998.

[30] Damien Pous. Relational algebra and KAT in Coq, February 2013. Available at `http://perso.ens-lyon.fr/damien.pous/ra`.

[31] Damien Pous. Symbolic algorithms for language equivalence and Kleene algebra with tests. In *POPL*, January 2015. To appear.

[32] Jurriaan Rot, Marcello M. Bonsangue, and Jan J. M. M. Rutten. Coalgebraic bisimulation-up-to. In *SOFSEM*, pages 369–381, January 2013.

[33] Jan J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In *CONCUR*, pages 194–218, September 1998.

[34] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.

[35] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *SIGCOMM*, pages 395–406, August 2014.

[36] Alexandra Silva. *Kleene Coalgebra*. PhD thesis, University of Nijmegen, 2010.

[37] Alexandra Silva. Position automata for Kleene algebra with tests. *Scientific Annals of Computer Science*, 22(2):367–394, 2012.

[38] Alexandra Silva, Filippo Bonchi, Marcello M. Bonsangue, and Jan J. M. M. Rutten. *Logical Methods in Computer Science*, 9(1):9, 2013.

[39] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, pages 235–249, January 2011.

[40] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, pages 87–98, August 2013.

[41] 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*, March 2005.

[42] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *CoNEXT*, pages 241–252, December 2012.