Temporal NetKAT

Ryan Beckett
Princeton University, USA
rbeckett@cs.princeton.edu

Michael Greenberg
Pomona College, USA
michael@cs.pomona.edu

David Walker
Princeton University, USA
dpw@cs.princeton.edu

Abstract

Over the past 5-10 years, the rise of software-defined networking (SDN) has inspired a wide range of new systems, libraries, hypervisors and languages for programming, monitoring, and debugging network behavior. Oftentimes, these systems are disjoint—one language for programming and another for verification, and yet another for run-time monitoring and debugging. In this paper, we present a new, unified framework, called Temporal NetKAT, capable of facilitating all of these tasks at once. As its name suggests, Temporal NetKAT is the synthesis of two formal theories: past-time (finite trace) linear temporal logic and (network) Kleene Algebra with Tests. Temporal predicates allow programmers to write down concise properties of a packet's *path* through the network and to make dynamic packet-forwarding, access control or debugging decisions on that basis. In addition to being useful for programming, the combined equational theory of LTL and NetKAT facilitates proofs of path-based correctness properties. Using new, general, proof techniques, we show that the equational semantics is sound with respect to the denotational semantics, and, for a class of programs we call *network-wide programs*, complete. We have also implemented a compiler for temporal NetKAT, evaluated its performance on a range of benchmarks, and studied the effectiveness of several optimizations.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

Keywords Software-defined networking, Network programming languages, Domain-specific languages, Kleene algebra with tests, NetKAT, Temporal logic.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4261-2/16/06...
http://dx.doi.org/10.1145/2908080.2908108

1. Introduction

In software-defined networking, a general-purpose controller machine, or cluster of machines, manages a collection of simple, programmable switches through a uniform and open API such as OpenFlow [29]. In order to build reliable SDN systems, one requires a combination of several key technologies: a platform for programming packet-forwarding policies, technology for monitoring dynamic traffic patterns and sampling packets of interest, and mechanisms for SDN verification. Over the past decade, all of these components have been tackled in a myriad of different ways. For instance, FlowLog [34], Frenetic [9], L [36], Maple [42], NetKAT [1], and others have given us effective new languages for programming packet-forwarding policies. DREAM [32], Open Sketch [43], Path Queries [33] and others provide sophisticated monitoring infrastructure. Header Space Analysis [17], NetPlumber [18], NetKAT again [10], Network Optimized Datalog [25] and Veriflow [19] are tools for checking various path properties of networks such as the absence of loops, access control, and waypointing.

In this paper, we present a simple, new foundation for managing many of these activities in the same framework. More specifically, we begin with NetKAT [1], which serves as a basis for modular programming of packet-forwarding functions. NetKAT programs consist of simple *predicates* that identify packets based on their current headers and location, and *policies* that modify those headers and forward packets through the network. A collection of combinators makes it possible to take the union of two policies, stitch policies together in sequence or iterate over a policy zero or more times. In combination, these features allow users to program network packet-processing functions.

However, as discussed above, network operators typically need additional support above and beyond facilities for programming packet-processing functions. In particular, many network monitoring and debugging tasks are most easily phrased in terms of *packet history*—the path and possible state changes a packet took to arrive at the current location. For instance, users might want to write queries that ask where specific traffic causing congestion is coming from or whether any packets headed towards a secure host have circumvented a network firewall. Each such question requires we understand the paths (or history) of packets as they flow

through a network. Moreover, the last question, in particular, suggests one might want to do something important depending on the answer: If a packet is headed towards my secure host and did circumvent the firewall, then I may want to reroute it through my deep-packet inspection box. While the above questions are *dynamic*—they ask about packets or traffic volume—*static* questions, which ask about properties of the routes (independent of the actual packets that might be flowing over them right now) are equally important. For instance, one might want to know given the current forwarding policy, whether any packet can travel from point A to B without going through a particular waypoint.

It is well known that temporal logic can also be used to specify static properties of networks such as connectivity and waypointing. Indeed, it has been used in verification of data plane properties [13, 38] and to specify invariants that must be preserved during network update [28]. In this paper we show that, when combined with the NetKAT language, it can be used for other tasks as well, such as network debugging, history-based routing and network monitoring. A crucial observation is that temporal logic is strong enough for all of the applications we have investigated and yet weak enough so that it retains the sound and complete equational theory akin to that of the original NetKAT language [1].

We make use of this observation to integrate the modal fragment of linear time temporal logic in to NetKAT in a deep way, by extending its language of predicates with the past-time temporal modalities "last," "ever," and "always," calling the resulting system *Temporal NetKAT*. With these modalities, rather than simply writing predicates that can inspect the current state of the packet, programmers can write conditions that refer to past states of a packet—where it was, what middleboxes it went through, or what the original state of its headers were prior to modification by NATs. Such predicates can be used to write concise programs that redirect packets to servers for logging, monitoring, traffic engineering or debugging purposes.

We have implemented a compiler for temporal NetKAT that converts temporal formula in to automata and then intersects these automata with standard NetKAT automata representing the rest of the policy. The combined automaton is then compiled to standard OpenFlow rules using BDDs, following work on the fast NetKAT compiler [39]. Along the way, we introduce new optimizations for improving compile times, reducing the number of rules generated and minimizing the tag space required.

To summarize the central contributions of this paper:

1. We introduce the design, applications and semantics of *Temporal NetKAT*, a simple, uniform language in which users can engage in network programming, monitoring, debugging and verification. (See Section 2 for an overview and Section 3.2 for the formal definitions.)
2. We develop an equational theory for Temporal NetKAT by combining NetKAT axioms, an original set of axioms for LTL_f, and new axioms relating actions and time.
3. We prove that, with respect to the denotational semantics, this new equational theory is both sound and complete for "network-wide" programs – programs where packet history begins at network entry. Though our focus is on networking, Kleene algebra and temporal logic have applications in other domains. We have developed a *generic* meta-theory so our core results may be reused in other settings where KAT embeds a richer notion of test. (See Section 3.3 for the equational theory and Section 4 for a proof outline; complete proofs appear in the appendix.)
4. We have implemented, optimized and studied the performance of a Temporal NetKAT compiler on a range of benchmarks (See section 5 for details). Our specialized compiler optimizations reduce compile times by orders of magnitude in some cases, allowing temporal policies to compile in seconds on the Stanford University Campus network, while also reducing rule set sizes by 20-50%.

2. Overview

In this section, we give an informal overview of Temporal NetKAT, beginning with an overview of NetKAT itself.

NetKAT. NetKAT, the predecessor of Temporal NetKAT, allows programmers to specify network policies using a small set of primitive commands and a collection of combinators. As a first cut, each NetKAT policy may be thought of as a function that takes a *located packet* as an input and returns a set of located packets as an output. These located packets are records that include the packet's headers and its location. The specific packet headers chosen are unimportant in this paper, but our examples will often include headers such as source IP (src), destination IP (dst), and "packet type" (typ — ssh or http perhaps). A packet's location is described in terms of its current switch (sw) and port (pt).

The simplest kind of NetKAT policy is a predicate that tests a packet's current contents and location. If the test is true, the policy returns a singleton set containing the packet unchanged. Otherwise, the policy returns the empty set, indicating the packet should be dropped. For example,

$$\mathsf{sw} = X \wedge \mathsf{pt} = 3 \wedge (\mathsf{typ} = \mathsf{ssh} \vee \mathsf{typ} = \mathsf{http})$$

is true when a packet is currently at port 3 on switch X, and the "typ" field of the packet header is either ssh or http. Such predicates restrict any policy that follows it to operate on certain sets of packets—here, SSH and HTTP traffic on port 3 of switch X. The always-false predicate (0) drops all packets; and the always-true predicate (1) admits all packets.

More complex policies not only read packet state but actively *modify* it. For example, the policy $\mathsf{src} \leftarrow 10.0.0.1$ updates the src field of any packet to have value 10.0.0.1. We also implement packet forwarding using assignment. For

example, to move a packet across the switch fabric to port 7, we write $\mathsf{pt} \leftarrow 7$. Predicates and primitive actions are combined to form more interesting policies using sequential composition of policies (;), parallel union of policies (+), and policy iteration (*). For example, to define the behavior of a switch X that sends SSH traffic out port 1 and all other traffic out port 2, we would use the following policy:

$$\begin{aligned} &((\mathsf{sw} = X \wedge \mathsf{typ} = \mathsf{ssh}); \mathsf{pt} \leftarrow 1) \\ +\ &((\mathsf{sw} = X \wedge \neg\mathsf{typ} = \mathsf{ssh}); \mathsf{pt} \leftarrow 2) \end{aligned}$$

The predicates $(\mathsf{sw} = X \wedge \mathsf{typ} = \mathsf{ssh})$ and $(\mathsf{sw} = X \wedge \neg\mathsf{typ} = \mathsf{ssh})$ identify subsets of the incoming switch traffic and the appropriate forwarding behavior (sending the packet out port 1 or port 2, respectively) is applied to each subset.

Now, it turns out that NetKAT can also be given a more refined semantics, one that transforms entire *packet histories* in to sets of *packet histories*. These packet histories are non-empty lists of packets that not only represent a packet's current state, but also prior states that occurred as it traversed the network. For instance, a history $pk_1{:}pk_2{:}pk_3{:}[\,]$ indicates that a packet is currently in the state pk_1, was in the state pk_2 before that and was in the state pk_3 before that.

This history-based semantics makes it possible for a network operator to examine a policy and reason about the path that a packet takes through a network. However, all such analysis occurs in the NetKAT metatheory. Even though NetKAT's semantics is in terms of packet histories, NetKAT programs themselves have no means to inspect or take action based on packet history: predicates are only allowed to inspect the *current* packet and packet rewriting actions only affect the current packet state. The only NetKAT operator that has any effect on packet history other than the current packet is the special dup action, which adds an extra copy of a packet to the front of the history and indicates an observable state change has occurred. For instance, if a history h is a sequence of past packet states with pk the current packet, then the semantics of dup can be explained as follows.

$$\mathsf{dup}(pk{:}h) = \{pk{:}pk{:}h\}$$

Temporal NetKAT. Temporal NetKAT extends NetKAT by enriching NetKAT's sublanguage of predicates with the temporal operators $\bigcirc\, a$ ("last a"), $\Diamond\, a$ ("ever a"), $\Box\, a$ ("always a"), and $a\ \mathcal{S}\ b$ ("a since b"). Each of these operators is interpreted over a packet's history as it traverses the network. Intuitively, these predicates have the following semantics: $\bigcirc\, a$ is true now, if a is true in the prior packet state, $\Diamond\, a$ is true now, if a is true now *or* it was true at *some* point in the past, $\Box\, a$ is true now if a is true now *and* it was true at *all* points in the past, and finally, $a\ \mathcal{S}\ b$ is true now if b was true at some previous state, and a has been true in *every* state since b held.

In the case that no prior packet state exists, what should be the semantics of $\bigcirc\, a$? A common trick for applying LTL to finite domains is to extend the domain to be infinite, e.g., by repeating the initial packet state of a history forever. However, in doing so, we lose the ability to reason about the beginning of the packet's history. We adopt the convention of LTL_f (LTL over finite traces [6, 7]) and say $\bigcirc\, a$ is false when no prior history exists. Under this semantics, in Temporal NetKAT, we can define a special predicate $\mathsf{start} \stackrel{\mathsf{def}}{=} \neg \bigcirc 1$ that holds precisely when a packet first enters the network, and has no previous history. It should also be noted that, although we use the $\Box$ and $\Diamond$ operators extensively throughout the rest of the examples in this section, they are in fact defined in terms of the $\mathcal{S}$ operator (see Section 3.2).

In regular NetKAT, the programmer can filter packets based on their current state; in Temporal NetKAT, the programmer can filter packets based on past states. Such facilities are particularly useful in network debugging. For example, one might want to look at packets that were accidentally routed around the network firewall. In this example, let's assume the firewall is implemented at switches F_1 and F_2 and a vulnerable host at switch V. Let's also assume the cont policy routes packets to the SDN controller for analysis and debugging. To capture packets that evade the firewall, we might use the following query.

$$\mathsf{q}_1 \stackrel{\mathsf{def}}{=} \neg \Diamond(\mathsf{sw} = F_1 \vee \mathsf{sw} = F_2) \wedge \mathsf{sw} = V; \mathsf{cont}$$

In English, it reads "if a packet has never passed through switch F_1 or switch F_2 and has arrived at switch V, then send it to the controller." Of course, if the vulnerable hosts attached to V send packets as well as receive them, we might want to exclude those packets that begin their journey through the network at switch V from our query. We may do that by conjoining "¬start" as follows.

$$\mathsf{q}_2 \stackrel{\mathsf{def}}{=} \neg \Diamond(\mathsf{sw} = F_1 \vee \mathsf{sw} = F_2) \wedge \mathsf{sw} = V \wedge \neg\mathsf{start}; \mathsf{cont}$$

Temporal NetKAT, like ordinary NetKAT, is highly compositional. Hence, to use either debugging query in parallel with *any* other routing program prog, we simply construct the parallel composition (+) of the query and the program, $\mathsf{prog} + \mathsf{q_i}$, which both routes packets to their destination, as dictated by prog, and also directs a copy of any packets that match the query to the controller.

Narayana et al. [33] have also observed that path-based queries are useful for a number of traffic engineering tasks—tasks that involve collection of traffic statistics for later route optimization. For example, a standard traffic matrix represents the traffic volume that flows between each network ingress i and network egress j over a given time period. In Temporal NetKAT, assuming the action $\mathsf{collector}(i,j)$ forwards traffic volume statistics to the traffic engineering application,[1] and predicates $\mathsf{ingress}_i$ and egress_j identify the

[1] Such an action may be implemented in exactly the way as Pyretic's "buckets" are [31].

Example	Predicate	Goal
Simple path	$\diamondsuit((\mathsf{sw} = S_1)@(\mathsf{sw} = S_2)@(\mathsf{sw} = S_3))$	Packets that travelled from S_1 to S_2 to S_3
Slice iso.	$\diamondsuit(\mathsf{slice}_1@\mathsf{slice}_2 \vee \mathsf{slice}_2@\mathsf{slice}_1)$	Packets travelling from slice_1 to slice_2 or vice-versa
Physical iso.	$\Box(\mathsf{sw} = S_1 \vee \ldots \vee \mathsf{sw} = S_n)$	Packets that only traverse switches S_1 through S_n
DDoS sources	$\sum_{S_i}(\mathsf{start} \wedge \mathsf{sw} = S_i)@\mathsf{server}$	Packets from switches S_i reaching a victim server
Congested link	$\diamondsuit((\mathsf{sw} = X \wedge \bigcirc(\mathsf{sw} = Y)) \vee (\mathsf{sw} = Y \wedge \bigcirc(\mathsf{sw} = X)))$	Packets using a congested link between switches X and Y

Figure 1: Example queries inspired by Narayana [33]. We only give the temporal predicates; to generate a query, the given predicates should be composed with the appropriate NetKAT actions ("forward to collector," "collect stats," *etc.*).

i^{th} and j^{th} ingress and egress locations respectively, we can set up the appropriate temporal query as follows.

$$\begin{aligned} &\mathsf{q}_3 \stackrel{\mathsf{def}}{=} \\ &\quad \diamondsuit(\mathsf{start} \wedge \mathsf{ingress}_1) \wedge \mathsf{egress}_1 \wedge \neg\mathsf{start}; \mathsf{collector}(1,1) \\ &\quad \ldots \\ &\quad \diamondsuit(\mathsf{start} \wedge \mathsf{ingress}_\mathsf{i}) \wedge \mathsf{egress}_\mathsf{j} \wedge \neg\mathsf{start}; \mathsf{collector}(i,j) \end{aligned}$$

To make formula such as this slightly easier to read, we can define a simple abbreviation for paths: $a@b$ stands for $(\diamondsuit\, a) \wedge b$ and @ is left associative so $a@b@c$ is $\diamondsuit((\diamondsuit\, a) \wedge b) \wedge c$. Hence to travel from $\mathsf{ingress}_1$ (entering the network) to egress_1 (not entering the network), we write: $(\mathsf{start} \wedge \mathsf{ingress}_1)@(\mathsf{egress}_1 \wedge \neg\mathsf{start})$. Narayana et al. [33] give a variety of other examples of path-based queries for debugging and monitoring. We present a selection of these additional examples in Figure 1, written as Temporal NetKAT queries.

In addition to network monitoring and debugging with Temporal NetKAT, programmers can, of course, make ordinary routing decisions based on packet history—the basic temporal mechanisms are the same, only the actions differ. For example, suppose packets coming from *badsrc* are untrustworthy and must be subjected to deep packet inspection (DPI) before being forwarded on to secure machines at the network edge. Due to the presence of network translation devices (NAT), such properties as "originated at *badsrc*" are not always apparent at the network edge or on the switches that make the final forwarding decision. However, history-based forwarding allows us to express desired behaviors accurately at a high level of abstraction. For instance, suppose we want to forward packets from *badsrc* that have gone through the middlebox out port 1 and other packets out port 2. We can formulate such a routing decision with the following temporal NetKAT policy.

$$\diamondsuit(badsrc@dpi); \mathsf{pt} \leftarrow 1 \ + \neg\diamondsuit(badsrc@dpi); \mathsf{pt} \leftarrow 2$$

Reasoning about packet histories. Verification of important network properties like reachability, waypointing, and loop-freedom can be formulated as problems of policy equivalence in the equational theory of NetKAT. To begin, consider a NetKAT user policy p operating in a network with topology t. In this case, the complete network policy can be formulated as the NetKAT expression:

$$\text{prog} \stackrel{\mathsf{def}}{=} \mathsf{dup}; (p; t; \mathsf{dup})^*$$

Now, if a network analyst wants to prove that all network traffic satisfies a given property, say that traffic traverses a series of middleboxes m_1 and m_2, they can begin their analysis by formulating their condition as the following temporal formula, which states that m_1 precedes m_2 in the past:

$$\text{query} \stackrel{\mathsf{def}}{=} \diamondsuit(m_1@m_2)$$

Now, we can simply ask whether all our network traffic traverses middlebox m with the following NetKAT equation:

$$\text{prog} \equiv \text{prog}; \text{query}$$

This decomposition of policy and query is highly modular. For example, if instead the operator wanted to ensure that all traffic goes through a NAT, which rewrites the source IP to x, then they could simply replace the query with:

$$\text{query} \stackrel{\mathsf{def}}{=} (\mathsf{src} = x)\ \mathcal{S}\ (\mathsf{sw} = \text{NAT})$$

Such questions can also be posed in pure NetKAT, without temporal operators, but doing so requires "mixing" the property of interest into the program to be verified. For instance, we might say that $\text{prog}' \stackrel{\mathsf{def}}{=} (p; t; \mathsf{dup})^*$. To verify the waypointing property, we prove that $\text{prog} \leq \text{prog}; m_1; \text{prog}'; m_2; \text{prog}'$, where $\leq$ is defined such that $a \leq b$ is equivalent to $a + b \equiv b$. *Why* this is right thing to prove isn't at all obvious, and must itself be proved [1]. As the properties become more complex, the degree of interleaving of property and program increases in NetKAT. Our temporal operators help modularize these specifications and keep proofs simple.

Summary. Temporal NetKAT facilitates modular debugging, querying, programming and verification of properties of packet history, all in the same uniform language, and it does so by extending the NetKAT predicates with past-time LTL_f.

3. Temporal NetKAT

Temporal NetKAT enriches the NetKAT language by embedding past-time LTL_f into its predicate sublanguage. Adding LTL_f operators starts a cascade of changes: to the semantics, to the equational theory, and to the metatheory. For reference, Figure 2 presents the combined syntax, denotational semantics and equational theory of Temporal NetKAT. The temporal extensions are highlighted in duckling. The details are explained below.

3.1 Preliminaries

A packet, written *pk*, is a record with a fixed number of abstract, fixed-width fields $f_1, \ldots, f_n$. In addition to conventional networking header fields—IP and Ethernet fields like source and destination, VLAN identifiers, etc.—our packets have virtual fields representing the packet's current location at a given switch (sw) or port (pt). We write $pk.f$ to denote the value in field f of packet pk, and we write $pk[f := v]$ to update field f with value v in packet pk.

A *packet history* (h) is a non-empty list of packets. For example, $pk_1{:}pk_2{:}\cdots{:}pk_n{:}[\,]$ is a history. Here, the left-most packet (pk_1) is the most recent packet state; the right-most packet (pk_n) is the oldest packet state. Such histories record the evolution of a packet as it traverses the network, including the sequence of changes to a packet's location, and changes to the contents of its header fields. For instance, a history may record changes to a packet's destination MAC address as it traverses a switch implementing Ethernet bridging. We call a history with one packet, $pk{:}[\,]$, an *initial history*—it represents a packet that has just entered our network. We often use the pattern "$pk{:}\cdots$" to bind pk to the current packet (*i.e.,* the most recent packet state) in a history, ignoring that packet's past states.

3.2 Syntax and Semantics

Temporal NetKAT generally follows the structure of other Kleene algebras with tests (KATs)—a Kleene algebra with an embedded Boolean algebra. The critical difference, of course, is the addition of the temporal operators from LTL_f.

Syntax. We break Temporal NetKAT's syntax (Figure 2, upper left) into two levels: *predicates* a and b (LTL_f expressions) and *policies* p and q (Kleene algebra terms). The stratified syntax ensures that negation and temporal operators are only applied to predicates, and not to policies.

The predicates include 0, pronounced "drop", the always-false predicate; and 1, pronounced "id", the always-true predicate. The field test $f = v$ tests whether a field has a given value. In the introductory sections of this paper, we wrote $a \wedge b$ and $a \vee b$ for conjunction and disjunction respectively to help ease the reader's eye. However, from this point forward, we adopt the convention of KAT and write $a; b$ for conjunction and $a + b$ for disjunction. The ; and + symbols have a single semantics, but may be used either to combine two predicates or to combine two policies (the latter being shown in the introductory section). We use $\neg a$ for negation. The two core temporal operators are $\bigcirc\, a$ (*last* a) and $a\ \mathcal{S}\ b$ (a *since* b). The operators $\Diamond\, a$ (*ever* a), $\Box\, a$ (*always* a), and $a\ \mathcal{B}\ b$ (a *back to* b) are derived forms, defined in terms of *last* and *since*.

We also introduce syntactic sugar to account for the finitude of LTL_f: start and $\bullet\, a$. The predicate start is defined as $\neg \bigcirc 1$, a predicate that only holds when packets first enter the network—that is, on initial histories. The *weak last* operator, $\bullet\, a$, is defined as $\neg \bigcirc(\neg a)$. It behaves just like $\bigcirc\, a$ except for on initial histories, where it is always true.

Every predicate is also a policy. Policies also include field assignments $f \leftarrow v$, which update the packet and lengthen the packet history to record the state change, and Kleene star p^*. As mentioned above, policies also reuse two of the predicate connectives: $p + q$ and $p; q$. When used as a policy, $p + q$ may be thought of as "parallel composition" (or the "union" policies) — it applies both p and q to any packet. The policy $p; q$ may be thought of as sequential composition of p and q. We let the unary operators ($\neg$, $\bigcirc$, $\Diamond$, $\Box$, and *) bind more tightly than (;), which binds more tightly than (+), which binds more tightly than $\mathcal{S}$ and $\mathcal{B}$.

Readers familiar with NetKAT will notice that we removed the dup operator from the syntax of policies. In Temporal NetKAT, we fold dup in to the semantics of the update operation. Hence, we automatically record every state change. Doing so helps slim down the notation used in our proofs, but it is not a change of any fundamental interest.

Semantics. We give a denotational semantics ($[\![-]\!]$ in Figure 2) defining Temporal NetKAT terms as functions from packet histories to sets of packet histories; the denotations are defined as a fixpoint on policies and histories. Intuitively, if $[\![p]\!]\ h = \emptyset$, then p drops its input packet. If it produces a singleton set, then p forwards a single copy of its input. If it produces a set of two or more packet histories, then p has *multicast* its input.

Given an input history, every predicate will either return the empty set of histories or a singleton set containing the input. For example, the denotation of 0 (false) is the empty set, while the denotation of 1 (true) is the singleton set containing the input packet. Field tests $f = v$ check the current, topmost packet *pk* of the input history to see whether $pk.f = v$, returning either the singleton set containing the input history (if they're equal) or the empty set (if not). Negation is defined using set difference, since predicates return either their inputs (as a singleton set) or the empty set. Field writes $f \leftarrow v$ extend the input history by copying the top-most packet and updating it: the new current packet is equal to the last current packet, but with field f set to v. Parallel composition $p + q$ (and disjunction) returns the union of the histories returned by p and q respectively. Sequential composition (and conjunction) returns the Kleisli composition (defined at the bottom-left of Figure 2) of p followed by q. Kleene star is defined as union of all finite sequences of a policy p.

Syntax

Predicates

a, b	$::=$	0	*drop*
	$\mid$	1	*id*
	$\mid$	$f = v$	*field test*
	$\mid$	$a; b$	*conjunction*
	$\mid$	$a + b$	*disjunction*
	$\mid$	$\neg a$	*negation*
	$\mid$	$\bigcirc\, a$	*last*
	$\mid$	$a\, \mathcal{S}\, b$	*since*

$\Diamond\, a = 1\, \mathcal{S}\, a$	*ever*
$\Box\, a = \neg \Diamond \neg a$	*always*
$\mathsf{start} = \neg \bigcirc 1$	*history start*
$\bullet\, a = \neg \bigcirc \neg a$	*weak last*
$(a\, \mathcal{B}\, b) = (a\, \mathcal{S}\, b) + \Box\, a$	*back to*

Policies

p, q	$::=$	a	*predicate*
	$\mid$	$f \leftarrow v$	*field update*
	$\mid$	$p; q$	*sequence*
	$\mid$	$p + q$	*parallel*
	$\mid$	p^*	*Kleene star*

Packets and packet histories

f	$::=$	$\{f_1, \ldots, f_n\}$	*fields*
V	$=$	$\{v_1, \ldots, v_m\}$	*values*
pk	$:$	$F \rightharpoonup V$	*packets*
h	$::=$	$pk{:}[\,]$	*pkt. history*
	$\mid$	$pk{:}h$	

Semantics

$$[\![p]\!] \;:\; \mathsf{History} \to 2^{\mathsf{History}}$$

$$\begin{aligned}
[\![0]\!]\, h &= \emptyset \\
[\![1]\!]\, h &= \{h\} \\
[\![f = v]\!]\, h &= \{h \mid h = pk{:}\ldots \wedge pk.f = v\} \\
[\![\neg a]\!]\, h &= \{h\} \setminus ([\![a]\!]\, h) \\
[\![f \leftarrow v]\!]\, h &= \{pk[f := v]{:}h \mid h = pk{:}\ldots\} \\
[\![p + q]\!]\, h &= ([\![p]\!]\, h) \cup ([\![q]\!] h) \\
[\![p; q]\!]\, h &= ([\![p]\!] \cdot [\![q]\!])\, h \\
[\![p^*]\!]\, h &= \textstyle\bigcup_{i \in \mathbb{N}} ([\![p]\!]^i\, h)
\end{aligned}$$

$$\begin{aligned}
[\![\bigcirc\, a]\!]\, pk{:}h &= \{pk{:}h \mid [\![a]\!]\, h = \{h\}\} \\
[\![a\, \mathcal{S}\, b]\!]\, pk{:}h &= ([\![b]\!]\, pk{:}h) \cup \\
&\quad ([\![a]\!]\, pk{:}h \cap \\
&\quad\quad \{pk{:}h \mid [\![a\, \mathcal{S}\, b]\!]\, h = \{h\}\})
\end{aligned}$$

$$\begin{aligned}
[\![\bigcirc\, a]\!]\, pk{:}[\,] &= \emptyset \\
[\![a\, \mathcal{S}\, b]\!]\, pk{:}[\,] &= [\![b]\!]\, pk{:}[\,]
\end{aligned}$$

$$\begin{aligned}
(F \cdot G)\, h &= \textstyle\bigcup_{h' \in F\, h} (G\, h') \\
F^0 &= [\![1]\!] \\
F^{i+1} &= (F \cdot F^i)
\end{aligned}$$

Equational theory

Kleene Algebra

$p + (q + r) \equiv (p + q) + r$	*KA-PLUS-ASSOC*
$p + q \equiv q + p$	*KA-PLUS-COMM*
$p + 0 \equiv p$	*KA-PLUS-ZERO*
$p + p \equiv p$	*KA-PLUS-IDEM*
$p; (q; r) \equiv (p; q); r$	*KA-SEQ-ASSOC*
$1; p \equiv p$	*KA-SEQ-ONE*
$p; 1 \equiv p$	*KA-ONE-SEQ*
$p; (q + r) \equiv p; q + p; r$	*KA-DIST-LEFT*
$(p + q); r \equiv p; r + q; r$	*KA-DIST-RIGHT*
$0; p \equiv 0$	*KA-ZERO-SEQ*
$p; 0 \equiv 0$	*KA-SEQ-ZERO*
$1 + p; p* \equiv p*$	*KA-UNROLL-LEFT*
$1 + p*; p \equiv p*$	*KA-UNROLL-RIGHT*
$q + p; r \leq r \;\to\; p*; q \leq r$	*KA-LFP-L*
$p + q; r \leq q \;\to\; p; r* \leq q$	*KA-LFP-R*

$$p \leq q \Leftrightarrow p + q = q$$

Boolean Algebra

$a + (b; c) \equiv (a + b); (a + c)$	*BA-PLUS-DIST*
$a + 1 \equiv 1$	*BA-PLUS-ONE*
$a + \neg a \equiv 1$	*BA-EXCL-MID*
$a; b \equiv b; a$	*BA-SEQ-COMM*
$a; \neg a \equiv 0$	*BA-CONTRA*
$a; a \equiv a$	*BA-SEQ-IDEM*

Linear Temporal Logic

$\bigcirc(a; b) \equiv \bigcirc\, a; \bigcirc\, b$	*LTL-LAST-DIST-SEQ*
$\bigcirc(a + b) \equiv \bigcirc\, a + \bigcirc\, b$	*LTL-LAST-DIST-PLUS*
$\bullet\, 1 \equiv 1$	*LTL-WLAST-ONE*
$a\, \mathcal{S}\, b \equiv b + a; \bigcirc(a\, \mathcal{S}\, b)$	*LTL-SINCE-UNROLL*
$\neg(a\, \mathcal{S}\, b) \equiv (\neg b)\, \mathcal{B}\, (\neg a; \neg b)$	*LTL-NOT-SINCE*
$a \leq \bullet\, a; b \;\to\; a \leq \Box\, b$	*LTL-INDUCTION*
$\Box\, a \leq \Diamond(\mathsf{start}; a)$	*LTL-FINITE*

Packet Axioms

$f \leftarrow v; \mathsf{start} \equiv 0$	*PA-MOD-START*
$f \leftarrow v; \bigcirc\, a \equiv a; f \leftarrow v$	*PA-MOD-LAST*
$f \leftarrow v; f' = v' \equiv f' = v'; f \leftarrow v,\ \text{if } f \neq f'$	*PA-MOD-COMM*
$f \leftarrow v; f = v \equiv f \leftarrow v$	*PA-MOD-FILTER*
$f = v; f = v' \equiv 0,\ \text{if } v \neq v'$	*PA-CONTRA*
$\sum_v f = v \equiv 1$	*PA-MATCH-ALL*

Useful Consequences

$\bullet\, a \equiv \mathsf{start} + \bigcirc\, a$	*WLAST-START*
$\neg \bigcirc\, a \equiv \mathsf{start} + \bigcirc \neg a$	*NOT-LAST*
$\Diamond\, a \equiv a + \bigcirc \Diamond\, a$	*EVER-UNROLL*
$\Box\, a \equiv a; \bullet \Box\, a$	*ALW-UNROLL*
$\mathsf{start}; \bullet\, a \equiv \mathsf{start}$	*START-WLAST*
$\mathsf{start}; \bigcirc\, a \equiv 0$	*START-LAST*
$\mathsf{start}; (a\, \mathcal{S}\, b) \equiv \mathsf{start}; b$	*START-SINCE*
$\mathsf{start}; (a\, \mathcal{B}\, b) \equiv \mathsf{start}; (a + b)$	*START-BACK*
$(p + q)^* \equiv q^*; (p; q^*)^*$	*KA-DENEST*
$p; (q; p)^* \equiv (p; q)^*; p$	*KA-SLIDING*
$p; a \equiv a; q + r \;\to\; p^*; a \equiv (a + p^*; r); q^*$	*TN-INVARIANT*
$p; a \equiv a; q + r \;\to\; p; a; (p; a)^* \equiv (a; q + r); (q + r)^*$	*TN-EXPAND*

Figure 2: Temporal NetKAT syntax, semantics, and equational theory (extensions of NetKAT are highlighted in duckling)

Temporal connectives are predicates, so their denotations must return either their input history as a singleton set or the empty set. We split our definitions into two cases: the inductive case, where the input history has length two or more, and the initial history case of pk:[ ].

To determine whether $\bigcirc a$ holds on $pk{:}h$ (where h is a history — a *non-empty* list of packets), we simply check whether a holds on h. We use set builder notation: $\{pk{:}h \mid [\![a]\!]\ h = \{h\}\}$ to capture this idea. Since a is a predicate, if a holds on h, then it returns the singleton set: $\{h\}$, and therefore $\bigcirc a$ returns the singleton set $\{pk{:}h\}$. Otherwise, if a does not hold on h, it will return the empty set, and thus the denotation of $\bigcirc a$ will result in the empty set as well. Finally, $\bigcirc a$ simply does not hold on an initial history $pk{:}[\,]$, since there was no previous time step.

We define $[\![a\ \mathcal{S}\ b]\!]\ pk{:}h$ by unrolling the temporal connective: It is the union of $[\![b]\!]\ pk{:}h$, which characterizes whether b holds now, and the intersection of $[\![a]\!]\ pk{:}h$ with $\{pk{:}h \mid [\![a\ \mathcal{S}\ b]\!]\ h = \{h\}\}$, which characterizes whether a holds now and $(a\ \mathcal{S}\ b)$ holds in the previous time step. For initial histories, $(a\ \mathcal{S}\ b)$ holds if and only if b holds *now*. This is *a crucial property* for reducing Temporal NetKAT to NetKAT: It suggests that if we can rewrite Temporal NetKAT policies so that the temporal operators appear only at network entry, we may eliminate them.

Because the semantics of $(a\ \mathcal{S}\ b)$ is such that b must be true at some point in the past, we can define the modal operator $\Diamond a$ as $(1\ \mathcal{S}\ a)$. This ensures that a holds at some point in the past, but is otherwise unconstrained. The other modal operator, $\Box$, is dual to $\Diamond$; we define $\Box a \stackrel{\text{def}}{=} \neg \Diamond \neg a \stackrel{\text{def}}{=} \neg(1\ \mathcal{S}\ \neg a)$, ensuring that a holds at every time step up to the present.

The LTL_f semantics also has the useful property that a program can detect the entry point of a packet to the network. The formula start $(\neg \bigcirc 1)$ is true with respect to h (*i.e.,* returns a non-empty set) if $\bigcirc 1$ is false, and $\bigcirc 1$ is false only if the history h is initial — in any other case, 1 will be satisfied by the prior state. This reasoning justifies our decision to define start as $\neg \bigcirc 1$. If a programmer wished to log all packets entering their network (by applying the "log" action), they need merely write $\mathsf{start}; \mathsf{log}$.

Since $\bigcirc$ is "brittle"—it fails at packet entry—it is useful to define a weak last operator $\bullet a$ as $\neg \bigcirc(\neg a)$ that succeeds at packet entry but otherwise acts as $\bigcirc a$. To see why, notice that $\bigcirc(\neg a)$ is false when a is true at the prior time step, or the history is initial. Consequently $\neg \bigcirc(\neg a)$ is true when a is true at the prior time step or the history is initial. Similarly, the *since* operator $(a\ \mathcal{S}\ b)$ can be "brittle" in that it requires that b hold some time in the past. We define the *back to* operator $(a\ \mathcal{B}\ b)$, which acts like the *since* operator, but allows for the case that b never holds.

3.3 Equational Theory

Formally, a *Kleene algebra* (KA) forms an idempotent semiring $\langle 0, 1, +, ; \rangle$ with a closure operator * called Kleene star satisfying extra unfolding axioms (*KA-UNROLL-* and *KA-LFP-* in Figure 2). A *Kleene algebra with tests* (KAT) embeds a Boolean subalgebra $\langle 0, 1, +, ;, \neg \rangle$ into the KA, with appropriate distributivity axioms [21]. Like other algebraic structures, the laws for KATs are typically phrased in terms of sets of (in)equations that must hold.

KATs typically enjoy sound and complete equational theories—powerful reasoning tools. NetKAT is a KAT with three extra kinds of terms: tests $f = v$, in the Boolean algebra; assignments $f \leftarrow v$ and history markers dup in the Kleene algebra. Accordingly some *packet axioms* are necessary to reason about how these new terms interact [12]. NetKAT also has a sound and complete equational theory.

Our equational theory (Figure 2) is broken up into four parts, named with corresponding prefixes: Kleene algebra axioms (*KA-*), Boolean algebra axioms (*BA-*), past-time LTL_f axioms (*LTL-*), and packet axioms (*PA-*). The majority of these axioms are standard, and we refer the curious reader to Anderson et al. [1] for in depth explanation. Three axioms from NetKAT are dropped, as they no longer hold:

$$f = v; f \leftarrow v \equiv f = v \qquad \textit{PA-FILTER-MOD}$$
$$f \leftarrow v; f \leftarrow v' \equiv f \leftarrow v' \qquad \textit{PA-MOD-MOD}$$
$$f \leftarrow v; f' \leftarrow v' \equiv f' \leftarrow v'; f \leftarrow v, \text{ if } f \neq f' \qquad \textit{PA-MOD-MOD-COMM}$$

These axioms no longer hold because $f \leftarrow v$ does not just update a field—it extends its history, too. Some of the axioms are phrased as implications and using $\leq$. The partial order $p \leq q$ is defined to be $p + q \equiv q$.

Kleene algebra axioms. These axioms include the familiar semiring laws plus the axioms for Kleene star.

Temporal logic axioms. LTL axioms are usually given in terms of implications or as a logical inference system [4, 24, 27]. However, here we are interested in an equational presentation. As part of the Temporal NetKAT equational theory, we provide an axiomatization of LTL_f that is, to the best of our knowledge, completely new. The axioms for LTL_f listed in Figure 2 are inspired by the complete axiomatizations for modal LTL given in Kröger and Merz [24] and the discussion of LTL_f by de Giacomo et al. [6, 7]. The use of LTL_f enables reasoning directly about the beginning of history and any other equivalences that hold only in the finite domain. This distinction plays an important role in the completeness proof in Section 4.

The axioms *LTL-LAST-DIST-SEQ* and *LTL-LAST-DIST-PLUS* say that $\bigcirc$ distributes over products and sums. The *LTL-WLAST-ONE* axiom tells us that, if there is a next state, then 1 will be true. The *LTL-SINCE-UNROLL* axiom says we can check if $(a\ \mathcal{S}\ b)$ is true by checking if either b is true now, or a is true now and $a\ \mathcal{S}\ b$ is true in the previous state.

The axiom *LTL-NOT-SINCE* describes how the negation and $\mathcal{S}$ operators interact. Intuitively, it states that if $(a\ \mathcal{S}\ b)$ does not hold, then either $\neg b$ always holds, or $\neg b$ holds until some state where neither a nor b hold. The axiom *LTL-INDUCTION* provides a way for reasoning about properties that hold over the entire history. It says that if whenever a is true, b is true and a also holds in the previous state when it exists, then b must always be true.

The final LTL_f axiom *LTL-FINITE* is a single new axiom capturing the finiteness of packet histories. It says that if we know a always holds, then we know eventually we will be at the beginning of the history and a will hold. This single new axiom allows us to reason directly about the end of packet histories. For example, the predicates $\Box \Diamond a$, $\Diamond \Box a$, and $\Diamond(\mathsf{start}; a)$ are all provably equivalent over finite traces even though they are distinct in the infinite setting.

Packet axioms. Only two new axioms relating temporal operators to the rest of the NetKAT are needed. The axiom *PA-MOD-LAST* relates adding to the history $(f \leftarrow v)$ with going back in time to inspect the history $(\bigcirc a)$. The axiom *PA-MOD-START* tells us that we can not be at an initial history after we have added to history via a modification.

Consequences. Useful consequences that play a role in the completeness proof are also given in Figure 2. Several of these consequences tell us how to expand temporal operators in terms of $\bigcirc$ as well as how to remove negation in front of $\bigcirc$. This ends up being important for LTL_f, since the usual equivalence for the duality of negation in LTL: $\neg \bigcirc a \equiv \bigcirc \neg a$ no longer holds. The **-UNROLL* axioms show how to unroll the $\Diamond$ and $\Box$ operators a single step. The START-lemmas are also unique to LTL_f, and provide a way to remove temporal operators at the beginning of history.

Of particular importance are the consequences: *TN-INVARIANT* and *TN-EXPAND*. These consequences deal with equivalences of the form $p; a \equiv a; q + r$, a situation that often arises due to the increased expressiveness of the LTL_f predicate language. For example, $(f \leftarrow v); \Diamond a \equiv \Diamond a; (f \leftarrow v) + (f \leftarrow v; a)$. These equivalences show how to rewrite terms where temporal predicates appear in sequence with $(p^*; a)$, or nested under $(p; a)^*$ the Kleene star operator into terms where the temporal predicate appears only at the beginning. They are used for the most difficult cases of the completeness proof, which shows that such rewriting can always occur and will eventually terminate.

4. Metatheory

Our main theoretical results involve *soundness* and *completeness* of the equational theory with respect to the denotational semantics for network-wide programs. The proof of completeness develops a new normalization strategy for dealing with rich predicate languages embedded in a Kleene algebra. This strategy allows us to rewrite policies into a form that cleanly separates the temporal logic fragment from the Kleene algebra fragment. Importantly, our new technique applies to other Kleene algebra domains that embed non-standard tests.

4.1 Soundness

Temporal NetKAT is *sound* if the policies that are equivalent in the theory—$p \equiv q$—have equivalent denotations $[\![p]\!] = [\![q]\!]$—i.e., for all packet histories h, $[\![p]\!]\ h = [\![q]\!]\ h$. A full proof of soundness appears in the appendix.

Theorem 1 (Soundness)**.** *If $p \equiv q$ then $[\![p]\!] = [\![q]\!]$, i.e., for all histories h, $[\![p]\!]\ h = [\![q]\!]\ h$.*

4.2 Completeness

In this section, we prove that the Temporal NetKAT equational theory is complete with respect to its denotational semantics for *network-wide* programs. That is, we show that for all policies p and q, if $[\![\mathsf{start}; p]\!] = [\![\mathsf{start}; q]\!]$, then $\mathsf{start}; p \equiv \mathsf{start}; q$. Network-wide programs describe the class of networks where packets entering the network have no prior history — e.g., previous switch locations. In practice, network-wide programs are usually the only case we are concerned with since historical information about packets before entering the network is typically not available. The proof of completeness proceeds in three steps:

1. We show that every policy is provably equivalent to a policy in *negation free form*, where negation appears only implicitly in the start predicate.
2. We introduce a new normal form for Temporal NetKAT in which tests appear only at the beginning of policies and show that every negation-free Temporal NetKAT policy is provably equivalent to a policy in normal form.
3. Using the normal form, we show that every *network-wide* policy is provably equivalent to a *history-free* NetKAT policy in which no temporal operators appear. We derive completeness for Temporal NetKAT through the completeness proof for NetKAT.

Key Proof Idea. The normal form used for NetKAT is a sum of policies of the form $\alpha; x; \pi$, in which a single leading test over all fields α appears at the beginning of the policy, and a single assignment for all fields π appears at the end of the policy (with x also taking on a particular regular structure). Composition of normal forms in the NetKAT completeness proof, particularly in the sequence and Kleene star cases, is made easier by the fact that the leading tests of one normal form will either cancel or be absorbed by the trailing assignments of another. For example, consider the following normalized policies placed in sequence (simplified for a single field):

$$(\mathsf{sw} = S_1; x; \mathsf{sw} \leftarrow S_4); (\mathsf{sw} = S_4; y; \mathsf{sw} \leftarrow S_6)$$

Normalizing this sequence is easy because $\mathsf{sw} \leftarrow S_4; \mathsf{sw} = S_4$ is equivalent to $\mathsf{sw} \leftarrow S_4$ in NetKAT. This allows us to

remove the test $\mathsf{sw} = S_4$ from the middle of the expression (and the goal of normalization is to eliminate tests appearing in the middle of the normal forms). Hence we are left with the following term, which can be re-normalized without having to reason about the structure of x and y:

$$\mathsf{sw} = S_1; (x; \mathsf{sw} \leftarrow S_4; y); \mathsf{sw} \leftarrow S_6$$

Moreover, such cancellation *always* happens in NetKAT. The presence of temporal operators makes normalization more challenging since we can no longer treat x and y as black boxes. Suppose instead we want to re-normalize two policies in sequence with an intervening temporal operator.

$$(\mathsf{sw} = S_1; x; \mathsf{sw} \leftarrow S_4); (\Diamond(\mathsf{sw} = S_2); y; \mathsf{sw} \leftarrow S_6)$$

The result will depend on x, which may be complex — potentially consisting of arbitrarily nested Kleene star operators. In particular, we will need to be able to move the test $\Diamond(\mathsf{sw} = S_2)$ through x to re-normalize our policy.

Note that, although we use operators such as $\Diamond$ for illustrative purposes here, the actual proof only deals with $\bigcirc$, $\mathcal{S}$, $\mathcal{B}$, and start. The main challenge is dealing with the competing fixpoint characteristics of $\mathcal{S}$ and $\mathcal{B}$ in the backwards direction for tests and Kleene star * in the forwards direction for policies. In fact, the fixpoint characteristic for $\mathcal{S}$ (and $\Diamond$, $\Box$) means that tests may actually grow in size as we push temporal operators back. For example:

$$\mathsf{sw} \leftarrow S_4; \Box \bigcirc a \equiv (\Box \bigcirc a; a); \mathsf{sw} \leftarrow S_4$$

A simple finite "unfolding" of the policy x is not enough. Furthermore, pulling temporal operators out from underneath a Kleene star operator is even more difficult. These challenges lead us to develop a notion of *maximal tests*, tests that cannot be generated by any other tests during normalization. Maximal tests combined with the *TN-* equivalences in Figure 2 will guarantee that we can continue to make progress towards normalizing a policy. Our main normalization proof proceeds using a non-standard inductive hypothesis based on maximal tests.

Step 1. As a first step, we translate Temporal NetKAT policies to negation free form by pushing negation symbols inward until they appear only implicitly as part of the start predicate. There is a straightforward conversion to negation free form for each LTL_f test, which is justified by the equational axioms and consequences. Figure 3 shows the syntax and translation to negated normal form for tests. We can make use of the packet axiom *PA-MATCH-ALL* to remove negation from a primitive test on a field $f = v$:

$$\neg(f = v) \equiv (\sum_{v'} f = v'); \neg(f = v) \equiv \sum_{v' \neq v} f = v'$$

The other cases follow either from Boolean algebra, or from the unfolding equivalences. The translation introduces the possible use of the start predicate and $\mathcal{B}$ operator. Because the rest of the completeness proof works on terms with tests in negated normal form, it thus always handles cases for start and $\mathcal{B}$ in addition to the usual temporal operators.

Negated normal form (nnf)

$a, b \ := \ \mathsf{start} \mid 0 \mid 1 \mid f = v \mid a; b \mid a + b \mid \bigcirc a \mid a \,\mathcal{S}\, b \mid a \,\mathcal{B}\, b$

Translation to nnf

$$\begin{array}{lcl}
nnf(\neg 0) & = & 1 \\
nnf(\neg 1) & = & 0 \\
nnf(\neg(f = v)) & = & \sum_{v' \neq v} f = v' \\
nnf(\neg\neg a) & = & nnf(a) \\
nnf(\neg(a + b)) & = & nnf(\neg a); nnf(\neg b) \\
nnf(\neg(a; b)) & = & nnf(\neg a) + nnf(\neg b) \\
nnf(\neg \bigcirc a) & = & \mathsf{start} + \bigcirc\, nnf(\neg a) \\
nnf(\neg(a \,\mathcal{S}\, b)) & = & nnf(\neg b) \,\mathcal{B}\, nnf(\neg(a + b))
\end{array}$$

Figure 3: Translation and syntax for negated normal form.

Step 2. To normalize policies in Temporal NetKAT, we show how to move an arbitrary LTL_f test through an arbitrary NetKAT program algebraically. We define a normal form, in which all LTL_f tests appear only at the beginning of policies, and show that every expression is provably equivalent to an expression in normal form.

Definition 1 (Normal forms)**.** *A policy p is a normal form if it is a sum of policies: $\sum_i a_i; m_i$, where each a_i is a test in nnf, and each m_i is either a test-free policy or* 1. *We say a policy p normalizes to q if $p \equiv q$ and q is a normal form.*

Theorem 2 (Normalization)**.** *Every Temporal NetKAT policy p normalizes to some normal form q such that $\vdash p \equiv q$.*

The proof relies on a helper lemma that shows normal forms can be re-normalized when placed in sequence or under a Kleene star. This proof is by induction on:

(i) an order on normal forms defined by maximal tests
(ii) the size of each normal form (# of AST nodes)

Step 3. The final step is to show how Temporal NetKAT can be reduced to NetKAT. The high-level idea is that, because we are concened with *network-wide* completeness, and because all temporal operators appear at the beginning of the normal form, we can apply the *START-** equational consequences to remove temporal operators. For example, $\mathsf{start}; ((\bigcirc a); p + (b \ \mathcal{S} \ c); q) \ \equiv \ \mathsf{start}; (a; p + c; q)$, which removes the temporal operators $\bigcirc$ and $\mathcal{S}$. This process can then be repeated inductively on a and c. The final resulting terms will be vanilla NetKAT terms, and although there are slight differences (e.g., our handling of the dup operator), the same idea for the completeness proof of NetKAT now applies. The final completeness theorem is:

Theorem 3 (Completeness)**.** *For all policies p and q, if* $[\![\mathsf{start}; p]\!] = [\![\mathsf{start}; q]\!]$ *then* $\vdash \mathsf{start}; p \equiv \mathsf{start}; q$.

Our normalization procedure and maximal tests are a general proof technique applicable beyond Temporal NetKAT. For example, we have proven it can be used to deal with more complex instructions, such as those that increment values (like those in P4 [5]), or more general tests, such as equality over numeric values.[2]

5. Compilation

In order to use Temporal NetKAT in real networks, we must compile high-level policies into low-level forwarding rules that can be installed on commodity network devices. As with previous work on NetKAT, we target the OpenFlow [29] standard for SDN, where forwarding rules consist of simple prioritized match-action pairs on packet headers. Smolka et al. [39] recently developed algorithms using symbolic automata for efficiently compiling arbitrary NetKAT policies, which uses a variant of BDDs called Forwarding decision diagrams (FDDs) to compactly represents the automaton's state transition and acceptance functions. The compilation stategy will extract switch-local policies from the automaton by tagging packets with the state of the automaton the packet is in as it traverses the network.

The challenge for Temporal NetKAT lies in making decisions based on a packet's history even though such information is not present in the actual packet. The main insight is that the same packet-tagging mechanism used in the NetKAT compiler can be used to encode information about pertinent temporal queries as well. In particular, we show how to compile Temporal NetKAT policies into NetKAT automata, thereby reusing existing compilation algorithms to implement dynamic query monitoring. Our compilation strategy builds on this previous work and extends it by:

1. Compiling LTL_f formulae into symbolic automata that track whether the query has been satisfied.
2. Replacing temporal predicates with abstract predicates representing the result of the query, and compiling this temporal-free policy into a symbolic automaton (thereby reusing existing NetKAT compilation techniques [39]).
3. Using a non-standard automata intersection operation to combine query and policy automata, while replacing abstract predicates with concrete values.
4. Performing a number of new optimizations.

5.1 Symbolic NetKAT Automata

Every NetKAT policy can be translated to an equivalent NetKAT automaton that matches a corresponding set of packet histories. Because NetKAT policies can both test fields as well as update them, each transition in a NetKAT automaton can be intuitively thought of as matching a pair of packets as a base character – one for the test, and another for the update. This idea is lifted to symbolic NetKAT automata [39] by letting the transition function describe sets of pairs of packets. More formally, a symbolic NetKAT automaton is defined as:

- S – a finite set of states
- $s_0 \in S$ – the initial state
- $\epsilon : S \to \mathsf{Pk} \to \mathcal{P}(\mathsf{Pk})$ – the acceptance function
- $\delta : S \to \mathsf{Pk} \to \mathcal{P}(\mathsf{Pk} \times S)$ – the transition function

The acceptance function maps a state and a packet to a set of output packets (i.e., the result of any packet modifications). For example, $\mathsf{sw} = X; \mathsf{pt} = 1; (\mathsf{pt} \leftarrow 2 + \mathsf{pt} \leftarrow 3)$ can be thought of as an acceptance function for state s of an automaton that maps packets at switch x and port 1 to a set of two output packets, one at port 2 and another at port 3 while mapping all other packets to the empty set. Similarly, the transition function maps a state and a packet to a set of new packets, each with a new state. We write the transition function on edges and the acceptance function in square brackets in Figure 4.

The transition and acceptance functions for NetKAT automata can be represented and manipulated efficiently using FDDs, which can be thought of as a kind of Multi-Terminal Binary Decision Diagram [11], where leaf nodes correspond to sets of packet assignments, and internal nodes correspond to tests on a particular packet field and value.

5.2 Compilation by Example

As a simple example to demonstrate compilation, suppose we want to send packets across two switches A and B so long as the source ip address was x at some point in the past. We identify packets entering the network with the ingress test $\mathsf{in} = (\mathsf{sw} = A; \mathsf{pt} = 1)$, and send them out port 2 with the assignment $\mathsf{pt} \leftarrow 2$. Next we define a topology $\mathsf{link} = (\mathsf{sw} \leftarrow B; \mathsf{pt} \leftarrow 1)$ that will move such packets across the link from switch A to switch B afterwards and port 1 afterwards. The basic policy for moving the packet across switches A and B is now:

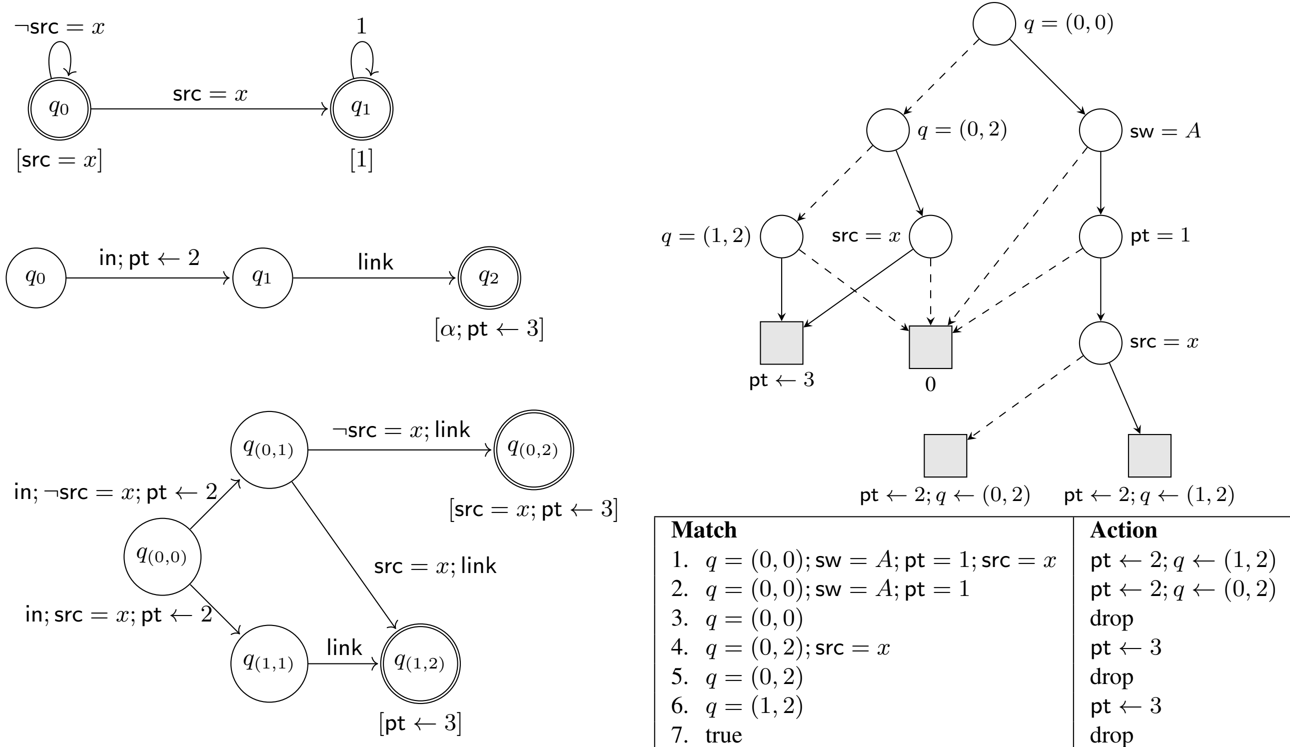
$$\mathsf{pol} \stackrel{\mathsf{def}}{=} \mathsf{in}; \mathsf{pt} \leftarrow 2; \mathsf{link}$$

Next, we add a temporal query that drops any packets where the source address has not been x at some point in the past, by filtering packets at switch B with the query: $\Diamond(\mathsf{src} = x)$. For those packets satisfying the query, we send the packet out port 3 on switch B by attaching the modification $\mathsf{pt} \leftarrow 3$ after the query. The final policy is now:

$$\mathsf{pol}' \stackrel{\mathsf{def}}{=} \mathsf{pol}; \Diamond(\mathsf{src} = x); \mathsf{pt} \leftarrow 3$$

Policy Automaton. The first step in our compilation process is to extract all of the queries from the Temporal NetKAT term, and replace each with a unique *abstract* placeholder variable. In the example above, we replace $\Diamond(\mathsf{src} = x)$ with variable α, resulting in $(\mathsf{pol}; \alpha; \mathsf{pt} \leftarrow 3)$.

[2] See appendix for additional discussion.



Match	Action
1. $q = (0, 0)$; sw $= A$; pt $= 1$; src $= x$	pt ← 2; $q \leftarrow (1, 2)$
2. $q = (0, 0)$; sw $= A$; pt $= 1$	pt ← 2; $q \leftarrow (0, 2)$
3. $q = (0, 0)$	drop
4. $q = (0, 2)$; src $= x$	pt ← 3
5. $q = (0, 2)$	drop
6. $q = (1, 2)$	pt ← 3
7. true	drop

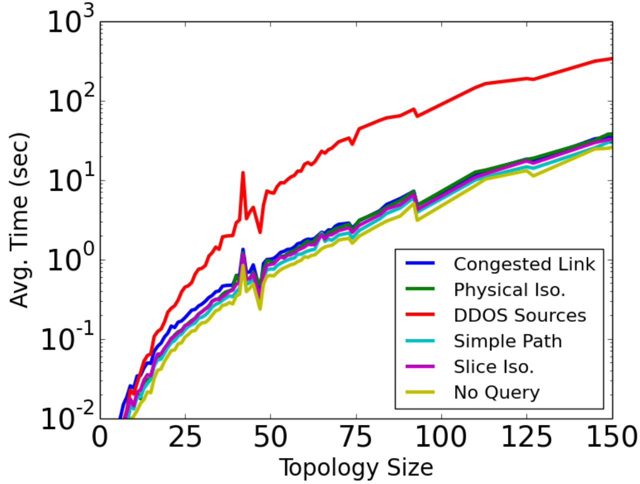
Figure 4: Query automaton (top left), Policy automaton (middle left), Product automaton (bottom left), Policy FDD (top right), forwarding rules (bottom right). Transition functions shown on edges. Acceptance functions in square brackets.

By treating α as abstract, we can compile the policy using existing symbolic NetKAT compilation techniques. Figure 4 (middle left) shows a representation of the automaton for the compiled NetKAT policy with the abstract variable α. In the automaton, packets that match the in predicate move to state q_1 of the automaton and the packet itself will be modified to move to port 2 of switch A. The transition from state q_1 to state q_2 will match any packet, and then move the packet to port 1 of switch B. Packets in state q_2 that satisfy the query α will move to port 3 and then be accepted by the automaton.

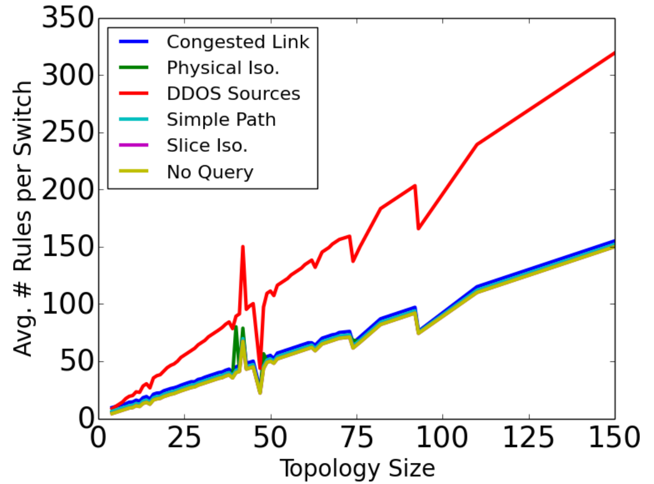
Query Automaton. The next step is to determine when the abstract variable α is satisfied. We compile each query that appears in the policy to a regular automaton that runs in the forwards direction of time, and tells us, at each point in time, what predicate must hold for the query to be satisfied (square brackets in Figure 4). Algorithms for compiling an LTL$_f$ term to an equivalent NFA have been explored by de Giacomo et al. [7]. We use a relatively simple translation from LTL$_f$ directly to an NFA that works fine in practice given the relatively small size of queries in proportion to the rest of the policy. To compile a test $f = v$, we create a single state that accepts when $f = v$ and transitions unconditionally to itself. To compile a test of the form: $\Diamond a$, we first compile the symbolic automaton for a, then introduce a new state that "remembers" if a has ever been satisfied. The new automaton for $\Diamond a$ will replace each acceptance function in the automaton for a with a transition to the new state, which will then accept unconditionally. The other operators plus $(+)$, $(;)$, and $(\neg)$ become automata union, intersection, and negation respectively. The case for $(a \ \mathcal{S} \ b)$ is more complicated, but follows the same idea. Using this construction, the final query automaton (after determinization) for $\Diamond(\mathsf{src} = x)$ is shown in Figure 4. As soon as a packet is observed with src $= x$, the query remains in an accepting state and accepts unconditionally.

Product Automaton. Finally, we combine the query and policy automata, resulting in a product automaton (Figure 4 bottom left) that uses information from the query automata to determine what each abstract variable should be. The acceptance function for the product automaton is just the acceptance function for the policy automaton, where the abstract placeholder variable is substituted for the acceptance function for the current state of the query automaton. For example, the acceptance function for state $q_{(0,2)}$ of the product automaton will be α; pt $\leftarrow$ 3 with α substituted with the acceptance function src $= x$ since we are in state q_0 of the query automaton. Likewise, the acceptance function for state $q_{(1,2)}$ will substitute 1 for α. The intuition is that the query automaton acceptance function tells us under what conditions the query is satisfied in any given state.

Transitions in the product graph are defined as follows: when there is a trasition function t_1 from state q_{x_1} to q_{x_2}



(a) Compilation time



(b) Number of forwarding rules

Figure 5: Compiler performance for shortest paths routing on the Topology Zoo

in the query automaton and a transition function t_2 from state q_{y_1} to q_{y_2} in the policy automaton, then there is a transition function $t_1; t_2$ from state $q_{(x_1,y_1)}$ to state $q_{(x_2,y_2)}$ in the product automaton. The resulting transition function $t_1; t_2$ represents the conjunction of the transition functions where any occurences of α are substituted with the concrete value from the query automaton's acceptance function.

From Automata to Forwarding Rules. The next step in compilation involves going from a network-wide NetKAT automaton to switch-local forwarding rules. This is accomplished by encoding the state of the automaton in a packet field (e.g., using an MPLS tag) so that each switch can simply see what automaton state the packet is in, and use that information to decide what forwarding action to take.

The idea is to extract a single FDD that encodes the forwarding actions of the entire network as shown in Figure 4 (top right). This FDD is obtained by taking the union $(+)$ of both the transition and acceptance functions of the combined automaton, where the state itself is encoded as a field in the FDD. One subtle difference is that the FDD does not encode "topology" states – i.e., states in the automaton that modify the switch, since such modifications happen due to physical network infrastructure rather than the switch hardware itself. For example, packets in state $q_{(0,0)}$ will be modifed to transition directly to state $q_{(0,2)}$ or state $q_{(1,2)}$. We refer the curious reader to Smolka et al. [39] for more information.

Network forwarding rules, e.g., like those of the OpenFlow standard can be thought of as simple prioritized match action rules on packet fields. Compilation from the FDD to network forwarding rules is straightforward – one simply traverses each path of the FDD, taking the conjunction of positive tests along the path. The final forwarding rules are shown in Figure 4 (bottom right).[3]

5.3 Optimizations

We use several optimizations to improve compile times and reduce rule overhead for Temporal NetKAT policies.

State-tag reduction. Often the forwarding rules will perform the same actions when in most states, and different actions when in a few other states. Whenever there is a collection of rules that differ only by the state they match, and collectively they match on all states, then we are free to reorder these rules. In particular, since forwarding rules are priority-based, we can select the largest subset of rules with the same forwarding action, and move them after the others. Since there is an implicit negation in the rule priority, we can drop the state test and compress them into a single rule. For example, consider the following rule optimization:

Match	**Action**
$\mathsf{sw} = A; \mathsf{dst} = x; q = 1$	$\mathsf{pt} \leftarrow 2; q \leftarrow 2$
$\mathsf{sw} = A; \mathsf{dst} = x; q = 2$	$\mathsf{pt} \leftarrow 1$
$\mathsf{sw} = A; \mathsf{dst} = x; q = 3$	$\mathsf{pt} \leftarrow 1$
$\mathsf{sw} = A; \mathsf{dst} = x; q = 4$	$\mathsf{pt} \leftarrow 1$
$\mathsf{sw} = A; \mathsf{dst} = x; q = 5$	$\mathsf{pt} \leftarrow 2; q \leftarrow 4$

Match	**Action**
$\mathsf{sw} = A; \mathsf{dst} = x; q = 1$	$\mathsf{pt} \leftarrow 2; q \leftarrow 2$
$\mathsf{sw} = A; \mathsf{dst} = x; q = 5$	$\mathsf{pt} \leftarrow 2; q \leftarrow 4$
$\mathsf{sw} = A; \mathsf{dst} = x$	$\mathsf{pt} \leftarrow 1$

If there are only 5 possible states, q_1 through q_5, then the above rules can be rearranged so the largest subset with the same action ($\mathsf{pt} \leftarrow 1$ in this case) are covered with an implicit negation. The resulting rules would be: Any packets

[3] We omit rules that tag packets entering the network with the initial state

Size	Time (sec)			Rules		
	unopt	opt	ratio	unopt	opt	ratio
5	.01	.01	1.0	102	52	.51
10	.02	.02	1.0	503	213	.42
15	.07	.06	.86	1354	484	.67
20	.23	.20	.87	3206	886	.28
25	.48	.40	.83	5632	1382	.25
30	1.14	1.03	.90	9008	1988	.22
35	2.52	1.33	.53	13484	2704	.20
40	12.41	2.93	.24	20811	3571	.17
45	48.66	4.51	.09	28362	4512	.16
50	245.89	6.70	.03	37513	5563	.15
55	T.O.	9.39	N/A	T.O.	6724	N/A
60	T.O.	15.41	N/A	T.O.	8056	N/A
65	T.O.	19.32	N/A	T.O.	9442	N/A
70	T.O.	30.97	N/A	T.O.	10938	N/A

Figure 6: Effect of optimizations on compilation time and rule overhead for the DDOS monitoring query with different sized topologies on the Topology Zoo. Timeouts after ten minutes shown with T.O.

that are not in state 1 or 5, but which match the same switch and destination in this case, will be caught by the final rule. This optimization is useful for only explicitly enumerating forwarding actions in states where the different forwarding action is meaningful.

Fall-through elimination. We use fall-through elimination, a rule-minimization technique [1] that eliminates rules that have no impact on forwarding behavior. For example, rules 3 and 5 from Figure 4 are unnecessary and can be safely removed since the default rule, which matches all packets, will drop the packets anyway.

Disjoint query check. Because query placeholder variables are abstract it is possible to experience a blowup in compilation time due to the inability of the BDD data structure to prune branches that can be eliminated. In practice however, many queries are disjoint. For example, in the DDoS sources query, each test assumed the packet entered from a unique ingress location in the network. Our compiler tests if queries are disjoint by checking if their intersection is empty. When a query, with the abstract variable α, is disjoint from another query, represented by β, we can encode the disjointness by replacing occurences of α with $\alpha \wedge \neg\beta$.

Partial Minimization. Because the size of the automata can have a large effect on both compilation time and, in particular, rule overhead, we perform partial minimization of the automata at each step of the compilation. Rather than performing full symbolic automata minimization, we observe that most of the unnecessary states arise from continuing to track the state of unsatisfiable queries. Our minimization merges states in the intersected automaton that will never reach an accepting state.

Query	Time (sec)			Rules		
	unopt	opt	ratio	unopt	opt	ratio
No Query	12.73	9.90	.78	3363	3363	1.0
Congested	13.81	11.77	.85	23538	4299	.18
Physical	12.28	10.83	.88	6757	3379	.50
DDOS	14.97	14.14	.94	23540	6363	.27
Simple	10.36	10.49	1.01	10088	4213	.42
Slice Iso.	12.80	9.97	.78	3363	3363	1.0

Figure 7: Effect of optimizations on compilation time and rule overhead per query on the Stanford campus network.

6. Evaluation

We have built a prototype compiler in OCaml for Temporal NetKAT that compiles network-wide policies down to forwarding rules – sequences of prioritized match-action rules. To evaluate our compiler, we measure its performance on several real-world examples, including the Stanford University network [16], and a large collection of real-world networks from the Topology Zoo [20]. For each network, we compile policies using variants of each of the queries listed in Figure 1 to monitor and collect traffic. For the DDoS sources example, since the number of queries can be variable, we fix the number of source locations from which we monitor traffic at N=20. We measure performance in terms of compilation time and forwarding rule overhead to determine whether temporal queries can (a) be compiled in a reasonable amount of time and (b) fit into switch memory. All experiments were run on a MacBook Pro with an 8-core, 2.4 GHz Intel Core i7 with 8GB RAM.

Topology Zoo. First, we evaluate our compiler on the Topology Zoo [20], a collection of hundreds of real-world network topologies. For each topology, we use a shortest paths routing policy that routes based on the packet's destination. Since each node in the network is associated with a unique destination, for a network with n switches, there will be on the order of n^2 total forwarding rules without queries.

Stanford Network. Next, we evaluate our compiler on the Stanford University network [16], a mid-sized campus network consisting of 16 backbone routers with hundreds of interfaces. The Stanford network policy is less uniform than the Topology Zoo examples and provides a useful point of comparison. When translated to Temporal NetKAT syntax, the policy totals 35676 AST nodes.

Compilation Time. Compilation time for networks of varying topology sizes in the Topology Zoo are shown in Figure 5a. The compiler scales to policies for large topologies in tens of seconds in most cases. For all queries except the DDOS sources query, the compiler adds very little overhead over the case where no queries exist. The most expensive query is the DDoS sources query, which not scale as well due to the large number of queries in the policy. Figure 7 shows compilation times for the Stanford University

campus network. All queries for the campus network compile in under 15 seconds in the worst case. Query overhead in the Stanford network over the case with no queries is in line with that of the Topology zoo as well and is never worse than a factor of 1.5 from the case without queries.

Rule Overhead. Figure 5b shows the effect of each query on the number of forwarding rules per switch in the Topology Zoo. For all queries except the DDOS sources query, only a small constant number of additional forwarding rules are needed to implement the query. In almost all of these cases, the rule overhead for the queries is near minimal and roughly what a human operator could reasonably be expected to write. The DDOS query on the other hand requires roughly 2x the number of forwarding rules. Although twice as many rules can still reasonably fit into switch memory in this case, this query is not optimal, and would likely benefit further from full automata minimization.

Optimization Effectiveness. The rule compression optimizations discussed in Section 5.3 are effective, greatly reducing the number of rules needed to implement the policy for many examples. Figure 6 shows the effect of our optimizations for Topology Zoo policies on varying sized topologies with the DDOS query. These optimizations result in a 50-85% reduction in the number of forwarding rules, and improve compilation time dramatically. Queries that time out on a topology of size 55 easily compile with optimizations. We also measure the effectiveness of our optimizations by query rather than topology size. Figure 7 shows the effects of our optimizations for each query on the fixed Stanford topology. Once again, the optimizations are effective, reducing the number of forwarding rules to a fraction of the original size.

7. Related Work

A number of new programming languages for SDN have been proposed recently including Maple [42], Flowlog [34], Nettle [41] and Merlin [40]. Our work on Temporal NetKAT is most closely related to NetKAT [1], one language in the Frenetic family of languages [9, 30, 31]. These languages do not include a notion of packet history that makes it easy to write queries based on where packets came from or what states they might have been in when first entering the network. L [36], is a declarative language for synthesizing network programs based on history. However, its notion of history is that of previous packets entering the network rather than the history of a single packet as it traverses the network.

Narayana et al [33] instrument the network with expressive Path Queries for collecting traffic statistics, monitoring, and debugging the network. While the current work was heavily inspired by Path Queries, Path Queries has no formal semantics. The main difference in design is that Path Queries are based on regular expressions as opposed to Temporal Logic, and our temporal predicates may be freely mixed with other NetKAT terms whereas Path Queries may not. We chose temporal logic because it seems to suit our application domain just as well as regular expressions, and yet we were able to develop a tractable equational theory for the extended language. We were deterred from pursuing an extension with a regular language that includes the useful intersection and negation operations, because of the apparent difficulty in developing a corresponding equational theory. Axiomatizations for Kleene Algebra, without tests, and with intersection but not negation are explored in Antimirov and Mosses [3] and Andréka et al. [2], but we do not know of a complete axiomatization for Kleene Algebra with Tests with intersection and negation.

The networking community has also developed a number of verification tools recently [17–19, 26]. For instance, the NetPlumber verification tool [18] includes a regular-expression-based specification language for describing path properties. Many of these properties (*e.g.,* packet waypointing) can be encoded as equivalences in Temporal NetKAT. The NetKAT language itself has also been used for verification, where program properties are framed as problems of program equivalence. For example, Foster *et al.* [10] develop a decision procedure that they use to verify properties on large, real-world networks. Temporal NetKAT retains important theoretical properties of NetKAT including its sound and complete equational theory while also admitting more modular specifications for verification.

Traditionally, LTL is interpreted over infinite traces, and has been used by programmers to specify properties of non-terminating programs. LTL over finite traces, LTL_f has become an object of study more recently [6, 7] due to its subtle differences with LTL — e.g., many formulas that hold in the finite case may no longer in the infinite case. Axiomatizations of LTL have been well-studied and are usually given as a logical inference system [4, 24, 27], however our equational axiomatization of LTL_f is, to the best of our knowledge, novel. Both LTL and LTL_f are traditionally endogenous languages — where the universe of discourse consists of a single, globally fixed program. In contrast, LTL_f is a first-class citizen of the language in Temporal NetKAT. It is freely mixed with NetKAT terms to direct network traffic.

Propositional Dynamic Logic (PDL) [15] subsumes KAT by adding modalities to programs of the form $[p]\phi$ meaning, after all executions of program p, ϕ holds. PDL reasons about programs by asking hypothetical questions about what is true after running a program, but it is not as well suited for reasoning about the ongoing behavior of programs. For example, it is difficult to describe properties such as $p; \Diamond(sw = X)$ — i.e., at some point in the execution of program p, the packet was at switch X. Modal Kleene Algebra [8] adds domain and codomain operators to KAT, yielding an algebraic alternative to PDL.

Process Logic [14, 35, 37], subsumes both temporal logic and PDL through the addition of temporal path formulas to

PDL of the form $[p]\chi$. Process Logic answers questions of the form: if program p runs, will formula χ hold throughout all or some executions. However, the semantics of Process Logic differs from that of Temporal NetKAT, as each temporal formula is associated with a particular program p. This makes it difficult to describe many Temporal NetKAT programs, since all temporal queries are embedded in the same monolithic program, not bound by any particular scope.

Completeness proofs for variants of Kleene algebra such as KAT [23] and NetKAT [1] generally involve defining a suitable language model to characterize the equational theory. Kozen and Mamouras [22] identify a general set of conditions for constructing free language models for Kleene algebra with additional equational premises, which is based on string rewriting systems. The normalization proof for Temporal NetKAT can also be viewed as a (terminating and confluent) string rewriting system, however, it fails to meet the sufficient conditions set forth by the authors since the system is not *well-behaved*, i.e., for each rewritting rule the term may in fact grow in size.

8. Conclusions

In this paper we introduce Temporal NetKAT, a new language and logic for reasoning about, and programming with history in networks. Temporal NetKAT builds on the semantic foundations of NetKAT to offer an appealing mix of linear temporal logic and (network) Kleene algebra with tests, with applications in network programming, verification, runtime monitoring, and debugging. We present a combined equational theory of LTL_f and NetKAT and prove that the equational theory is sound and complete with respect to a broad class of programs called network-wide programs. We have also implemented, optimized and evaluated a compiler for compiling temporal NetKAT programs.

Acknowledgments

Aarti Gupta provided valuable feedback and helped us find the appropriate automata. Steffen Smolka and Nate Foster gave helpful advice on NetKAT automata and compilation strategies. This work was supported in part by the NSF under grant CNS 1111520 and in part by a gift from CISCO. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *POPL*, January 2014.

[2] H. Andréka, S. Mikulás, and I. Németi. The equational theory of kleene lattices. *Theor. Comput. Sci.*, 412(52):7099–7108, 2011.

[3] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143:195–209, 1994.

[4] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer Publishing Company, Incorporated, 3rd edition, 2012. ISBN 1447141288, 9781447141280.

[5] P. Bosshart, D. Daly, M. Izzard, N. McKeown, J. Rexford, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. Programming protocol-independent packet processors. See `http://arxiv.org/abs/1312.1719`, December 2013.

[6] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 854–860, 2013.

[7] G. De Giacomo, R. D. Masellis, and M. Montali. Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 1027–1033, 2014.

[8] J. Desharnais, B. Mller, and G. Struth. Modal kleene algebra and applications – a survey, 2004.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, September 2011.

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

[11] M. Fujita, P. McGeer, and J.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2):149–169.

[12] N. B. B. Grathwohl, D. Kozen, and K. Mamouras. KAT + B! In *Proc. Joint Meeting of the 23rd EACSL Conf. Computer Science Logic (CSL 2014) and 29th ACM/IEEE Symp. Logic in Computer Science (LICS 2014)*, 2014.

[13] S. Gutz, A. Story, C. Schlesinger, and N. Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, 2012.

[14] D. Harel, D. Kozen, and R. Parikh. Process logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144 – 170, 1982. doi: http://dx.doi.org/10.1016/0022-0000(82)90003-4.

[15] D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0262082896.

[16] P. Kazemian. *Mini-Stanford*, 2012 (accessed July 27, 2015). URL `https://bitbucket.org/peymank/hassel-public/wiki/Mini-Stanford`.

[17] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[18] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[19] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[20] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. In *IEEE Journal on Selected Areas in Communications*, 2011.

[21] D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366 – 390, 1994. doi: http://dx.doi.org/10.1006/inco.1994.1037.

[22] D. Kozen and K. Mamouras. *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, chapter Kleene Algebra with Equations, pages 280–292. Springer Berlin Heidelberg, 2014.

[23] D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In *Proc. 10th Int. Workshop Computer Science Logic (CSL'96), volume 1258 of Lecture Notes in Computer Science*, pages 244–259, 1996.

[24] F. Kröger and S. Merz. *Temporal Logic and State Systems (Texts in Theoretical Computer Science. An EATCS Series)*. Springer Publishing Company, Incorporated, 1 edition, 2008. ISBN 3540674012, 9783540674016.

[25] N. Lopes, N. Bjorner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992. ISBN 0-387-97664-7.

[28] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient synthesis of network updates. In *PLDI*, pages 196–207, 2015.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computing Communications Review*, 38(2):69–74, 2008. doi: http://doi.acm.org/10.1145/1355734.1355746.

[30] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *POPL*, January 2012.

[31] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *NSDI*, April 2013.

[32] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: dynamic resource allocation for software-defined measurement. In *SIGCOMM*, 2014.

[33] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling path queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 207–222, 2016.

[34] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *HotSDN*, 2013.

[35] H. Nishimura. Descriptively complete process logic. *Acta Inf.*, 14(4):359–369, 1980. doi: 10.1007/BF00286492.

[36] O. Padon, N. Immerman, A. Karbyshev, O. Lahav, M. Sagiv, and S. Shoham. Decentralizing SDN policies. In *POPL*, 2015.

[37] V. R. Pratt. Process logic: Preliminary report. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1979.

[38] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[39] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha. A fast compiler for NetKAT. In *ICFP*, ICFP 2015, 2015.

[40] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. *CoRR*, abs/1407.1199, 2014.

[41] A. Voellmy and P. Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, 2011.

[42] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.

[43] M. Yu, L. Jose, and R. Miao. Software-defined traffic management with opensketch. In *NSDI*, 2013.