Temporal NetKAT

Ryan Beckett
Princeton University
rbeckett@princeton.edu

Michael Greenberg
Princeton University
mg19@cs.princeton.edu

David Walker
Princeton University
dpw@cs.princeton.edu

Abstract

Recent collaborations between networking and programming languages researchers have given rise to a new wave of high-level languages for specifying the behavior of networks. One such language is Anderson *et al.*'s NetKAT [1], a formal system that allows programmers to both construct packet-forwarding policies and to reason about them. NetKAT policies consist of tests on packet contents, actions to modify packet contents, and combinators to build more expressive policies from simpler ones.

In this talk, we will argue that NetKAT would benefit from new operators that allow NetKAT programs to analyze and make decisions based on a packet's history instead of just a packet's current state. More specifically, we will show how to extend the syntax and denotational semantics of NetKAT with several new temporal operators that act on packet history. We will explore the utility of these operators when it comes to constructing queries for monitoring networks, implementing history-based routing, and reasoning about network properties. We will also discuss our work-in-progress on the equational theory and compilation techniques for handling this new language.

1. Introduction

One of the most important, and perhaps underappreciated, outcomes of early work on Software-Defined Networking (SDN) was the development a very, very simple model of how networks of switches could be programmed and controlled. The simplicity of the model made network programming accessible to a broad range of researchers, including those in programming languages and verification. Hence, almost immediately, new, higher-level languages and programming abstractions for SDN control began to emerge. Such languages and abstractions have provided many new properties and guarantees *by construction*, conveniences and reasoning principles missing in the past.

One quadrant of the SDN language design space is inhabited by the Frenetic family languages [2], the most recent of which being NetKAT [1]. NetKAT programs consist of two main components: *predicates* and *policies*. The predicates test a packet's current contents and location. 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 the policy to operate on certain sets of packets—here, SSH and HTTP traffic on port 3 of switch X. Policies, on the other hand, *modify* a packet's current state. Primitive policies include operations to drop a packet (drop) or modify one of its field ($\mathsf{srcIP} \leftarrow 10.0.0.1$). The packet's current location is considered a (virtual) field, so moving a packet from place to place is also achieved via field 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 ($\cdot$), 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 NetKAT policy:

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

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.

The semantics of NetKAT is given in terms of *packet histories*, which are non-empty lists of packets (including both packet contents and location). More precisely, every NetKAT policy is defined as a function from a packet history and to a set of packet histories. For instance, consider policy p and history h. If $p(h)$ is the empty set, then a packet with history h is dropped by the policy. If $p(h)$ is $\{h_1\}$ then the packet is forwarded somewhere. If $p(h)$ is $\{h_1, h_2\}$ then the packet is multicast to two different destinations.

This history-based semantics makes it possible for an analyst to examine a policy and reason about the path that a packet takes through a network. For example, it is possible to use NetKAT semantics to determine whether policy p forces all packets to waypoint through a particular firewall switch. 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 as opposed to a past packet state; packet rewriting actions only affect the current packet state. Indeed, the only NetKAT operator that has any effect on packet history other than the current packet is the dup action, which adds an extra copy of a packet to the front of the history. For instance, if a history h is a sequence of packet states $\langle s_1, s_2, \ldots, s_n\rangle$, with s_1 the current packet, then the semantics of dup can be explained as follows.

$$\mathsf{dup}(\langle s_1, s_2, \ldots, s_n\rangle) = \{\langle s_1, s_1, s_2, \ldots, s_n\rangle\}$$

In this talk, we discuss our work-in-progress on extending NetKAT with new features for programming with and reasoning about packet histories. More specifically, we will discuss our efforts to extend the NetKAT predicate language with past-time temporal operators such as $\mathsf{last}(a)$, which asks if a is true at the previous step in the history, and $\mathsf{ever}(a)$, which asks if a was ever true at any point in the history. We will illustrate how these extensions (1) allow analysts to concisely and directly specify interesting history-based properties of networks, (2) allow programmers to define network routing or security policies in terms of packet history, and (3) allow network operators to express traffic queries in terms of packet history. We also describe our work to date on the semantics and meta-theory of these extensions to NetKAT, which we call *Temporal NetKAT*.

Reasoning about packet histories. Verification of important network properties like reachability and waypointing can be formulated as 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-wide policy can be formulated as the NetKAT expression prog.

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

Now, if a network analyst wants to prove that all network traffic traverses a series of middleboxes, say 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:

$$\text{middleboxes} \stackrel{\text{def}}{=} \mathsf{ever}(\mathsf{ever}(m_1) \wedge m_2)$$

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

$$\text{prog} = \text{prog} \cdot \text{middleboxes}$$

Such questions can also be posed in vanilla NetKAT, but doing so requires "mixing" the property of interest into the program to be verified. For instance, one might first define prog$'$:

$$\text{prog}' \stackrel{\text{def}}{=} (p \cdot t \cdot \mathsf{dup})^*$$

Now, without the temporal operators, we can verify the waypointing property by checking the following inequality :

$$\text{prog} \leq \text{prog} \cdot m_1 \cdot \text{prog}' \cdot m_2 \cdot \text{prog}'$$

The less than or equal operator ($\leq$) is defined such that $a \leq b$ is equivalent to $a + b = b$. As the properties become more complex, the degree of interleaving of property and program increases; our temporal operators facilitate modularization of these specifications.

History-based routing and security. From a programmer's perspective, the benefit of Temporal NetKAT is that programs can make routing decisions based on a packet's history. 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 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.

$$\mathsf{ever}(\mathsf{ever}(badsrc) \cdot dpi) \cdot \mathsf{pt} \leftarrow 1 +$$
$$\neg\mathsf{ever}(\mathsf{ever}(badsrc) \cdot dpi) \cdot \mathsf{pt} \leftarrow 2$$

That is, not only can Temporal NetKAT verify waypointing properties, as above, it can *declare* them. We believe that the equational theory for last and ever will allow us to rewrite history-sensitive policies to implementable, history-free policies.

History-based queries. Despite great strides made by SDN in improving network programmability, network monitoring remains difficult because SDN's current abstractions for monitoring are at a very low level, in the form of individual rule match and byte counts on the switches. Recently, Narayana et al. [3] have shown how to define a higher-level network query language that uses regular expressions to describe the paths that packets traverse through the network. We are exploring the use of temporal NetKAT to express similar kinds of queries and to provide rigorous semantics for Narayana et al.'s query language. For example, given an arbitrary program prog, Temporal Netkat makes it easy to detect any packets that evade a firewall F and send them to the controller (via the tocontroller action). The query we would write is:

$$\text{query} \stackrel{\text{def}}{=} \neg\mathsf{ever}(\mathsf{sw} = F) \cdot \mathsf{tocontroller}$$

and then applying that query to the program is trivial:

$$\text{prog} \cdot \text{query}$$

Narayana et al. also show how to answer other queries such as: how many packets go through middlebox X and middlebox Y right after each other in either order? Again, we can answer this query by appending the following test to an arbitrary program:

$$\mathsf{ever}(\mathsf{last}(\mathsf{sw} = X) \cdot \mathsf{sw} = Y \ + \ \mathsf{last}(\mathsf{sw} = Y) \cdot \mathsf{sw} = X)$$

We believe that last and ever can express a large subset of Narayana et al.'s queries within Temporal NetKAT itself.

Meta-theory. We have given a formal semantics to the last and ever temporal operators and have augmented the equational theory of NetKAT with several new axioms for reasoning about the temporal operators. We have proven the axioms sound with respect to the denotational semantics and are currently working to show completeness. Using the equational theory, we can rewrite a number of policies that contain temporal operators into traditional NetKAT that is temporal-operator-free. We are still exploring a full compilation algorithm.

Acknowledgments

This work was supported in part by the NSF under grant CNS 1111520. 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] 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.

[3] S. Narayana, J. Rexford, and D. Walker. Compiling path queries in software-defined networks. In *HotSDN*, pages 181–186, 2014.