An Algebraic Approach to Automatic Reasoning for NetKAT Based on its Operational Semantics*

Yuxin Deng, Min Zhang**, and Guoqing Lei**

Shanghai Key Laboratory of Trustworthy Computing, MOE International Joint Lab of Trustworthy Software, and International Research Center of Trustworthy Software, East China Normal University

Abstract. NetKAT is a network programming language with a solid mathematical foundation. In this paper, we present an operational semantics and show that it is sound and complete with respect to its original axiomatic semantics. We achieve automatic reasoning for NetKAT such as reachability analysis and model checking of temporal properties, by formalizing the operational semantics in an algebraic executable specification language called Maude. In addition, as NetKAT policies are normalizable, two policies are operationally equivalent if and only if they can be converted into the same normal form.We provide a formal way of reasoning about network properties by turning the equivalence checking problem of NetKAT policies into the normalization problem that can be automated in Maude.

Keywords: NetKAT, operational semantics, model checking, LTL, Maude

1 Introduction

In recent years, there has been exciting developent in the area of *software-defined networking* (SDN), where physically distributed switches are programmable and managed in a logically centralized way so as to effectively implement many applications such as traffic monitoring, access control, and intrusion detection. Several domain-specific languages for SDN have been proposed, e.g [3, 6, 9, 11, 14, 15], in order to have a high-level abstraction of network programs where it is more effective to specify, program and reason about the behaviour of networks. Among them, NetKAT [3] is a network programming language based on Kleene algebra with tests (KAT) [10]. The design of NetKAT was influenced by NetCore [10] and Pyretic [12], both of which originate from Frenetic [6]. Different from other languages, NetKAT has a solid mathematical foundation with a denotational semantics and an axiomatic semantics based on KAT. It has also been extended to the probabilistic setting [7].

* Partially supported by the National Natural Science Foundation of China (Grant No. 61672229, 61261130589, 61502171), Shanghai Municipal Natural Science Foundation (16ZR1409100), and ANR 12IS02001 PACE.

** Corresponding authors. Email addresses: zhangmin@sei.ecnu.edu.cn (Min Zhang), 51151500022@ecnu.cn (Guoqing Lei)

In this paper we present an operational semantics for NetKAT. The basic idea is to view the (global) state of a network as the set of all packets currently availabe in the network. Transitions between states are enabled by the execution of policies: the behaviour of a policy is to transform a given packet into a (possibly empty) set of packets, which leads to the change of state for the whole network. The operational semantics induces a natural equivalence on policies. Intuitively, two policies p and q are equivalent, if starting from any state Π, both policies can change Π into the same state Π'. We show that two policies are operationally equivalent if and only if they are provably equal according to the equational theory defined in [3]. In other words, our operational semantics is sound and complete with respect to the original axiomatic semantics of NetKAT.

In order to facilitate the reasoning about NetKAT programs, we formalize the operational semantics and the normalization theory of NetKAT in an algebraic formal reasoning system Maude [5]. The operational semantics is executable in Maude so we can search if a desired state is reachable from a starting state. The normalization theory of NetKAT tells us that all policies are normalizable, and two policies are operationally equivalent if and only if they can be converted into the same normal form. This gives rise to a formal way of reasoning about network properties by turning the equivalence checking problem of NetKAT policies into the normalization problem. More specifically, to check if two policies are equivalent, we first normalize them and check if their normal forms are the same. Both steps can be automated in Maude. For instance, in order to check the reachability from one switch to another in a network, we first define a high-level specification that is independent of the underlying network topology and a low-level implementation that describes a hop-by-hop routing from the source to the destination. Both the specification and the implementation are written as NetKAT policies. We then exploit our rewriting-based reasoning to check the (in)equivalence of the two policies. If Maude produces a positive answer, meaning that the two policies are equivalent, we know that the implementation conforms to the specification, thus the destination switch is indeed reachable from the source switch. Equivalence checking of policies are also useful for other applications such as proving compilation correctness and non-interference property of programs [3]. In addition, we combine the formalized operational semantics with Maude LTL model checker to verify temporal properties of packets traveling in a network. This differs from [4], which enriches the original NetKAT language with temporal predicates to specify properties of a packet's history. The current work considers a lightweight NetKAT that does not record any packet history, but we still achieve automatic reasoning about packet histories by using Maude.

The rest of this paper is structured as follows. In Section 2 we recall the syntax and the axiomatic semantics of NetKAT as given in [3]. In Section 3 we present an operational semantics and show that it is sound and complete with respect to its axiomatic semantics. In Section 4 we formalize the operational and axiomatic semantics of NetKAT in Maude. In Section 5 we use Maude to do reachability analysis, model checking of LTL properties, and equivalence checking of policies. In Section 6 we discuss our experiments. Finally, we conclude in Section 7.

Table 1: Syntax of NetKAT

Fields	$f ::= f_1 \mid \cdots \mid f_k$						
Packets	$\pi ::= \{f_1 = n_1, \cdots, f_k = n_k\}$						
Predicates	$a, b, c ::=$	$\mathsf{1}$	*Identity*	Policies	$p, q, r ::=$	a	*Filter*
	$\mid$	$\mathsf{0}$	*Drop*		$\mid$	$f \leftarrow n$	*Modification*
	$\mid$	$f = n$	*Match*		$\mid$	$p + q$	*Parallel composition*
	$\mid$	$a + b$	*Disjunction*		$\mid$	$p \cdot q$	*Sequential composition*
	$\mid$	$a \cdot b$	*Conjunction*		$\mid$	p^*	*Kleene star*
	$\mid$	$\neg a$	*Negation*				

2 NetKAT

We briefly review the syntax and the axiomatic semantics of NetKAT; see [3] for a more detailed exposition.

NetKAT is based on Kleene algebra with tests (KAT) [10], an algebra for reasoning about partial correctness of programs. KAT is Kleene algebra (KA), the algebra of regular expressions, augmented with Boolean tests. Formally, a KAT is a two-sorted structure $(K, B, +, \cdot, *, \neg, \mathsf{0}, \mathsf{1})$ such that

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

Elements of B and K are called *tests* and *actions* respectively; they are called predicates and policies in NetKAT.

A packet π is a record with fields $f_1, ..., f_k$ mapping to fixed-width integers $n_1, ..., n_k$, respectively. We assume that every packet contains the same fields, including two special fields for the switch (sw) and the port (pt) that identify the position of a packet in a global network. We write $\pi.f$ for the value in field f of π, and $\pi[n/f]$ for the packet obtained by updating field f of π by value n.

The syntax of NetKAT is given in Table 1. There are two categories of expressions: predicates (a, b, c) and policies (p, q, r). Predicates include true ($\mathsf{1}$) and false ($\mathsf{0}$), matches $(f = n)$, negation $(\neg a)$, disjunction $(a + b)$, and conjunction $(a \cdot b)$ operators. Policies include predicates, modifications $(f \leftarrow n)$, parallel $(p + q)$ and sequential $(p \cdot q)$ composition, and iteration (p^*). By convention, $(^*)$ binds tighter than $(\cdot)$, which binds tighter than $(+)$. The only and key difference from the original NetKAT presented in [3] is the absence of the dup operator. This operator is hardly used in practical network programming and is introduced mainly to facilitate the completeness proof of an axiomatic semantics with respect to a denotational semantics [3]. For easy formalization and reasoning in Maude, it seems more reasonable to drop this operator than to keep it.

The axiomatic semantics of NetKAT is displayed in Table 2, where $p \leq q$ is an abbreviation for $p + q \equiv q$. We write $\vdash p \equiv q$ if the equality $p \equiv q$ is derivable by

Table 2: Axioms of NetKAT

Kleene Algebra Axioms			
$p + (q + r) \equiv (p + q) + r$	KA-Plus-Assoc	$(p + q) \cdot r \equiv p \cdot r + q \cdot r$	KA-Seq-Dist-R
$p + q \equiv q + p$	KA-Plus-Comm	$0 \cdot p \equiv 0$	KA-Zero-Seq
$p + 0 \equiv p$	KA-Plus-Zero	$p \cdot 0 \equiv 0$	KA-Seq-Zero
$p + p \equiv p$	KA-Plus-Idem	$1 + p \cdot p^* \equiv p^*$	KA-Unroll-L
$p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r$	KA-Seq-Assoc	$q + p \cdot r \leq r \Rightarrow p^* \cdot q \leq r$	KA-Lfp-L
$1 \cdot p \equiv p$	KA-One-Seq	$1 + p^* \cdot p \equiv p^*$	KA-Unroll-R
$p \cdot 1 \equiv p$	KA-Seq-One	$q + r \cdot p \leq r \Rightarrow q \cdot p^* \leq r$	KA-Lfp-R
$p \cdot (q + r) \equiv p \cdot q + p \cdot r$	KA-Seq-Dist-L		

Additional Boolean Algebra Axioms			
$a + (b \cdot c) \equiv (a + b) \cdot (a + c)$	BA-Plus-Dist	$a \cdot b \equiv b \cdot a$	BA-Seq-Comm
$a + 1 \equiv 1$	BA-Plus-One	$a \cdot \neg a \equiv 0$	BA-Contra
$a + \neg a \equiv 1$	BA-Excl-Mid	$a \cdot a \equiv a$	BA-Seq-Idem

Packet Algebra Axioms	
$f \leftarrow n \cdot f' \leftarrow n' \equiv f' \leftarrow n' \cdot f \leftarrow n$, if $f \neq f'$	PA-Mod-Mod-Comm
$f \leftarrow n \cdot f' = n' \equiv f' = n' \cdot f \leftarrow n$, if $f \neq f'$	PA-Mod-Filter-Comm
$f \leftarrow n \cdot f = n \equiv f \leftarrow n$	PA-Mod-Filter
$f = n \cdot f \leftarrow n \equiv f = n$	PA-Filter-Mod
$f \leftarrow n \cdot f \leftarrow n' \equiv f \leftarrow n'$	PA-Mod-Mod
$f = n \cdot f = n' \equiv 0$, if $n \neq n'$	PA-Contra
$\sum_i f = i \equiv 1$	PA-Match-All

using the axioms in Table 2. A denotional semantics based on packet histories is shown to be sound and complete with respect to the axiomatic semantics in [3].

3 Operational Semantics

Below we give an operational semantics for NetKAT. We assume a global network that consists of a finite number of switches. Each switch has a finite number of ports. A *state* of the network is the set of all packets in the network. We denote by **S** the set of all possible states in the network, ranged over by Π.

Intuitively, the behaviour of a policy is to transform a given packet into a (possibly empty) set of packets. This can be described by an evaluation relation of the form $\langle p, \pi \rangle \rightarrow \Pi$, where p is a policy, π is a packet to be processed and Π is the set of packets obtained by applying p to π, as defined in Table 3. The evaluation relation can be lifted to the form $\langle p, \Pi \rangle \rightarrow \Pi'$, where both Π and Π' are sets of packets, according to the last rule in Table 3.

Table 3: Operational semantics of NetKAT

$$\frac{}{\langle \mathbf{1}, \pi \rangle \to \{\pi\}}\ [\text{Identity}] \qquad \frac{}{\langle \mathbf{0}, \pi \rangle \to \emptyset}\ [\text{Drop}] \qquad \frac{}{\langle f \leftarrow n, \pi \rangle \to \{\pi[n/f]\}}\ [\text{Modification}]$$

$$\frac{\pi.f = n}{\langle f = n, \pi \rangle \to \{\pi\}}\ [\text{Match-I}] \qquad \frac{\pi.f \neq n}{\langle f = n, \pi \rangle \to \emptyset}\ [\text{Match-II}] \qquad \frac{\langle a, \pi \rangle \to \Pi}{\langle \neg a, \pi \rangle \to \{\pi\} \backslash \Pi}\ [\text{Negation}]$$

$$\frac{\langle p, \pi \rangle \to \Pi_p \quad \langle q, \pi \rangle \to \Pi_q}{\langle p + q, \pi \rangle \to \Pi_p \cup \Pi_q}\ [\text{Parallel composition}] \qquad \frac{\forall i \in I : \langle p, \pi_i \rangle \to \Pi_i}{\langle p, \{\pi_i\}_{i \in I} \rangle \to \cup_{i \in I} \Pi_i}\ [\text{Packet set}]$$

$$\frac{\langle p, \pi \rangle \to \{\pi_i \mid i \in I\} \quad \forall i \in I : \langle q, \pi_i \rangle \to \Pi_i}{\langle p \cdot q, \pi \rangle \to \cup_{i \in I} \Pi_i}\ [\text{Sequential composition}]$$

$$\frac{\langle p^0, \pi \rangle \to \Pi_0 = \{\pi\} \quad \forall i \geq 0 : \langle p^{i+1}, \pi \rangle = \langle p \cdot p^i, \pi \rangle \to \Pi_{i+1}}{\langle p^*, \pi \rangle \to \cup_{i \geq 0} \Pi_i}\ [\text{Kleene star}]$$

A pair of the form $\langle p, \Pi \rangle$ represents a configuration from which it remains to execute by applying policy p to state Π. The execution may terminate in a final state, or may diverge and never yield a final state, because the rule for p^* potentially requires infinite computations. However, in practical applications, we often specify p in such a way that after finitely many iterations, the set Π_i will stabilize to be empty, thus we can terminate the computation when a sufficiently large bound is reached. We will see in Section 6 a concrete example where the length of the selected path between two nodes in a network actually gives a bound for the number of iterations.

The operational semantics immediately induces an equivalence on policies.

Definition 1. *Two policies are* operationally equivalent*, written $p \sim q$, if*

$$\forall \Pi, \Pi' \in \boldsymbol{S} : \langle p, \Pi \rangle \to \Pi' \Leftrightarrow \langle q, \Pi \rangle \to \Pi'.$$

If two policies p and q are provably equal by using the axioms in Table 2, then they are operationally equivalent.

Theorem 2 (Completeness). *If $\vdash p \equiv q$ then $p \sim q$.*

Proof. Let us first define a denotational semantics as follows.

$$\begin{aligned}
[\![p]\!] &\in \Pi \to \mathcal{P}(\Pi) & [\![\neg a]\!]\pi &:= \{\pi\} \backslash ([\![a]\!]\pi) \\
[\![\mathbf{1}]\!]\pi &:= \{\pi\} & [\![f \leftarrow n]\!]\pi &:= \{\pi[n/f]\} \\
[\![\mathbf{0}]\!]\pi &:= \emptyset & [\![p + q]\!]\pi &:= [\![p]\!]\pi \cup [\![q]\!]\pi \\
[\![p \cdot q]\!]\pi &:= ([\![p]\!] \bullet [\![q]\!])\pi & [\![p^*]\!]\pi &:= \textstyle\bigcup_{i \in \mathbb{N}} F^i \pi \\
[\![f = n]\!]\pi &:= \begin{cases} \{\pi\} & \text{if } \pi.f = n \\ \emptyset & \text{otherwise} \end{cases},
\end{aligned}$$

where $F^0\pi := \{\pi\}$ and $F^{i+1}\pi := (\llbracket p \rrbracket \bullet F^i)\pi$, and $\bullet$ is the Kleisli composition of functions of type $\Pi \to \mathcal{P}(\Pi)$ defined as:

$$(f \bullet g)(x) := \bigcup \{g(y) \mid y \in f(x)\}.$$

This is essentially the standard packet-history semantics for NetKAT [3] without the `dup` operator. In the absence of this operator, only the current packet is recorded and all its history is forgotten. By the soundness of the packet-history semantics, we know that

$$\text{if } \vdash p \equiv q \text{ then } \llbracket p \rrbracket = \llbracket q \rrbracket. \tag{1}$$

By a simple induction on the structure of policies, it is not difficult to show that $\llbracket p \rrbracket \pi = \Pi$ iff $\langle p, \pi \rangle \to \Pi$. It follows that

$$\llbracket p \rrbracket = \llbracket q \rrbracket \text{ iff } p \sim q. \tag{2}$$

Combining (1) and (2), we obtain that $\vdash p \equiv q$ implies $p \sim q$. □

The inverse of the above theorem also holds, and the rest of this section is devoted to proving it. Inspired by [3] we first introduce a notion of reduced NetKAT in order to define normal forms of policies.

Let $f_1, ..., f_k$ be a list of all fields of a packet in some fixed order. For each tuple $\bar{n} = n_1, ..., n_k$ of values, let $\bar{f} = \bar{n}$ and $\bar{f} \leftarrow \bar{n}$ denote the expressions

$$f_1 = n_1 \cdot \ldots \cdot f_k = n_k \qquad\qquad f_1 \leftarrow n_1 \cdot \ldots \cdot f_k \leftarrow n_k,$$

respectively. The former is a predicate called an *atom* and the latter a policy called a *complete assignment*. The atoms and the complete assignments are in one-to-one correspondence according to the values $\bar{n}$. If α is an atom, we denote by σ_α the corresponding complete assignment, and if σ is a complete assignment, we denote by α_σ the corresponding atom. We write At and P for the sets of atoms and complete assignments, respectively. Note that all matches can be replaced by atoms and all modifications by complete assignments. Hence, any NetKAT policy may be viewed as a regular expression over the alphabet $At \cup P$.

Definition 3. *A policy is in* normal form *if it is in the form* $\sum_{i \in I} \alpha_i \cdot \sigma_i$*, where* I *is a finite set,* $\alpha_i \in At$ *and* $\sigma_i \in P$*. It degenerates into* `0` *if* I *is empty. A policy* p *is* normalizable *if* $\vdash p \equiv p'$ *for some* p' *in normal form. A normal form* p *is* uniform *if all the summands have the same atom* α*, that is* $p = \sum_{i \in I} \alpha \cdot \sigma_i$*.*

Lemma 4. *Every policy is normalizable.*

Proof. Similar to the normalization proof in [3]. The most difficult case is Kleene star. In order to obtain the normal form of p^*, we first need to consider the case that p is a uniform normal form, based on which we consider the general form and make use of an important KAT theorem called `KAT-Denesting` in [10]:

$$(p + q)^* \equiv p^* \cdot (q \cdot p^*)^* \ .$$

□

Theorem 5 (Soundness). *If $p \sim q$ then $\vdash p \equiv q$.*

Proof. By Lemma 4, we know that there are normal forms $\hat{p}$ and $\hat{q}$ such that $\vdash p \equiv \hat{p}$ and $\vdash q \equiv \hat{q}$. By completeness we have $p \sim \hat{p}$ and $q \sim \hat{q}$, which implies $\hat{p} \sim \hat{q}$ by transitivity. Let $\hat{p} = \sum_{i \in I} \alpha_i \cdot \sigma_i$ and $\hat{q} = \sum_{j \in J} \beta_j \cdot \rho_j$. Note that for each atom α there is a unique packet that satisfies it (if α is $\bar{f} = \bar{n}$, then the packet has fields $f_i = n_i$ for each $1 \leq i \leq k$). Let us denote this packet by π_α. The behaviour of any summand $\alpha_i \cdot \sigma_i$ is to block all packets that do not satisfy α_i and transform the packet π_{α_i} into $\pi_{\alpha_{\sigma_i}}$. In view of `KA-Plus-Idem`, we may assume that no two summands in $\hat{p}$ are the same; similarly for $\hat{q}$.

Below we infer from

$$\sum_{i \in I} \alpha_i \cdot \sigma_i \;\sim\; \sum_{j \in J} \beta_j \cdot \rho_j \tag{3}$$

that I is in one-to-one correspondence with J and

$$\forall i \in I, \exists j \in J : \; \alpha_i \cdot \sigma_i = \beta_j \cdot \rho_j. \tag{4}$$

To see this, take any packet π. By the operational semantics of NetKAT, we know that

$$\langle \sum_{i \in I} \alpha_i \cdot \sigma_i,\ \pi \rangle \to \bigcup_{i \in I} \Pi_i,$$

where $\Pi_i = \{\pi_{\alpha_{\sigma_i}}\}$ if $\pi = \pi_{\alpha_i}$, and $\emptyset$ otherwise. Similarly,

$$\langle \sum_{j \in J} \beta_j \cdot \rho_j,\ \pi \rangle \to \bigcup_{j \in J} \Pi'_j,$$

where $\Pi'_j = \{\pi_{\alpha_{\rho_j}}\}$ if $\pi = \pi_{\beta_j}$, and $\emptyset$ otherwise. We know from (3) that

$$\bigcup_{i \in I} \Pi_i = \bigcup_{j \in J} \Pi'_j. \tag{5}$$

If indeed $\pi = \pi_{\alpha_k}$ for some $k \in I$, we let $[k]_1$ be the set $\{i \in I \mid \alpha_k = \alpha_i\}$ and $[k]_2$ be the set $\{j \in J \mid \alpha_k = \beta_j\}$. We have that

$$\begin{array}{c} \bigcup_{i \in I} \Pi_i = \bigcup_{i \in [k]_1} \Pi_i = \bigcup_{i \in [k]_1} \{\pi_{\alpha_{\sigma_i}}\} \\ \bigcup_{j \in J} \Pi'_i = \bigcup_{j \in [k]_2} \Pi'_i = \bigcup_{j \in [k]_2} \{\pi_{\alpha_{\rho_j}}\}. \end{array}$$

Combining them with (5), we obtain that

$$\bigcup_{i \in [k]_1} \{\pi_{\alpha_{\sigma_i}}\} \;=\; \bigcup_{j \in [k]_2} \{\pi_{\alpha_{\rho_j}}\}.$$

Note that the elements in the left union are pair-wise different and similarly for the elements in the right union. Therefore, $[k]_1$ is in one-to-one correspondence with $[k]_2$, that is, for each $i \in [k]_1$ there is a unique $j \in [k]_2$ such that $\pi_{\alpha_{\sigma_i}} = \pi_{\alpha_{\rho_j}}$.

Observe that $\{[k]_1 \mid k \in I\}$ is actually a partition of I, and so is $\{[k]_2 \mid k \in I\}$ for J (there is no $j \in J$ with $\beta_j \neq \alpha_i$ for all $i \in I$, otherwise the packet π_{β_j} would be blocked by $\hat{p}$ but not by $\hat{q}$). This means that I is in one-to-one correspondence with J and for each $i \in I$ there is a corresponding $j \in J$ with $\alpha_i = \beta_j$ and $\pi_{\alpha_{\sigma_i}} = \pi_{\alpha_{\rho_j}}$. Note that the only complete assignment (a string in P) that produces $\pi_{\alpha_{\sigma_i}}$ is σ_i. So we must have $\sigma_i = \rho_j$ and hence $\alpha_i \cdot \sigma_i = \beta_j \cdot \rho_j$. Therefore, we have completed the proof of (4).

As a consequence, we can derive $\vdash \sum_{i \in I} \alpha_i \cdot \sigma_i \equiv \sum_{j \in J} \beta_j \cdot \rho_j$ by using `KA-Plus-Comm`, and hence $\vdash p \equiv q$ by transitivity. □

The proof of Lemma 4 is largely influenced by [3]. However, due to the absence of the `dup` operator, our proof of Theorem 5 is much simpler and more elementary than its counterpart [3, Theorem 2]; the latter is based on a reduction to the completeness of Kleene algebra, which is not needed any more in our proof.

4 Formalization of NetKAT in Maude

4.1 Maude in a nutshell

Maude is a state-of-the-art algebraic specification language and an efficient rewrite engine [5], which can be used to formally define semantics of programming languages. One main feature of Maude is that formal definitions in Maude are executable [13], which allows us to *execute* programs with the defined semantics and perform formal analysis for the programs.

Maude specifies both equational theories and rewrite theories. An equational theory is a pair $(\Sigma, E \cup A)$, where Σ is a signature specifying the sorts and operators, E is a set of equations, and A is a set of equational attributes. An equation is an unoriented pair of two terms t_1, t_2 of the same sort. In Maude, it is defined in the form of (`eq` t_1 `=` t_2`.`). An equation can be conditional, and it is defined in the form of (`ceq` t_1 `=` t_2 `if` c`.`), where c can be an ordinary equation $t = t'$, a matching equation $t := t'$, or a conjunction of such equations. A matching equation, e.g., $t := t'$, is mathematically treated as an ordinary equation, but operationally t is matched against the canonical form of t' and the new variables in t are instantiated by the matching. Although an equation is unoriented mathematically, they are used only from left to right by Maude for computation. Equations must be guaranteed terminating and confluent when they are used as simplification rules. Intuitively, terminating means that there must not exist an infinite sequence of applying these equations, and confluence means that the final result after applying these equations must be unique.

A rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$ consists of an underlying equational theory $(\Sigma, E \cup A)$ and a set of (possibly conditional) rewrite rules R. A rewrite rule is an oriented pair (from left to right), which is defined in the form of (`rl` t_1 `=>` t_2`.`) for the case of unconditional rules or (`crl` t_1 `=>` t_2 `if` c'`.`) for the case of conditional rules, where c' is a more general condition than that in conditional equations by allowing rewrite condition in the form of t `=>` t'. A rewrite condition t `=>` t' holds if and only if there exists a finite rewrite

sequence from t to t' by applying the rewrite rules in R. Computationally, both equations and rewrite rules are used from left to right to *rewrite* target terms. Mathematically, equations are interpreted as the definition of functions, while rewrite rules are interpreted as transitions or inference rules. Unlike equations, rewrite rules are not necessarily terminating and confluent.

Rewrite theories can be used to naturally specify transition systems or logical frameworks. The underlying equational theory is used to specify the statics of systems such as data types, state structures, and R specifies the dynamics, i.e., the transitions among states. System states are specified as elements of an algebraic data type, namely, the initial algebra of the equational theory (Σ, E). In Σ state constructors are declared to build up distributed states out of simpler state components. The equations in E specify the algebraic identities that such distributed states enjoy. The rewrite rules in R specify the local concurrent transitions of transition systems.

4.2 Formalization of the operational semantics of NetKAT

Before formalizing the operational semantics of NetKAT, we need first formalize the basic concepts such as fields, packets, policies and configuration in NetKAT. Maude allows for user-defined data types. We explain the definition of some important data types such as `Field`, `Packet`, `Policy` and `Configuration`.

```
sorts   FieldId, Field, Policy, Predicate, Configuration .
subsort Field < Packet .
ops src typ dst vlan ip-src ip-dst tcp-src tcp-dst udp-src
   udp-dst sw pt : -> FieldId [ctor] .
op (_:_)  : FieldId Int    -> Field  [ctor] .
op nil    :                -> Packet [ctor] .
op __     : Packet Packet  -> Packet [assoc ctor id: nil] .
op _←_    : FieldId Int    -> Policy [ctor] .
op _+_    : Policy Policy  -> Policy [ctor assoc comm] .
op _·_    : Policy Policy  -> Policy [ctor assoc] .
op _*     : Policy         -> Policy [ctor] .
ops 1 0   :                -> Predicate [ctor] .
op _=_    : FieldId Int    -> Predicate [ctor] .
op _+_ : Predicate Predicate -> Predicate [ctor assoc comm].
op _#_ : Predicate Predicate -> Predicate [ctor assoc comm].
op ~_  : Predicate           -> Predicate [ctor] .
op <_,_>  : Policy PackSet   -> Configuration .
```

The Maude keyword `sorts` is used to declare sorts to represent sets of data elements, and `subsort` declares a partial order relation of two sorts. By declaring that `Field` is a subsort of `Packet`, it formalizes the fact that a field is also regarded as a packet, but not vice versa. Keyword `op` (resp. `ops`) is used to declare an operator (resp. multiple operators). Maude allows infix operators, in which the underbars indicate the place where arguments should be located. Operator `(_:_)` is used to construct fields with field identifiers and integer numbers. Operator `nil` is called a constant because it does not take any arguments, and it represents

an empty packet. The union of two packets constitute a new one, as formalized by the operator `__`. The operators declared for policies and predicates have clear correspondence to the syntax defined in Table 1, and thus we omit more detailed explanations about them. It is worth mentioning that `ctor`, `assoc` and `comm` are attributions of operators, declaring that an operator is a constructor, associative and commutative, respectively. We use `o` to represent *Drop* and `l` for *Identity*. We declare a new operator `#` instead of $\cdot$ to represent conjunction of predicates because $\cdot$ is used for sequential composition of policies and is not commutative, while conjunction of predicates is commutative.

We declare a sort `Configration` to represent the sets of the pairs of the form $\langle p, \Pi \rangle$. An element of sort `Configuration` is called a *configuration*, written in the form of `<p,PI>` with a policy `p` and a set `PI` of packets.

The operational semantics of NetKAT is formalized by the transformation of a configuration into another. As defined in Table 3, the *execution* of a policy $p \cdot q$ can be viewed as a sequential execution of p and q. We define the following set of rewrite rules with each formalizing one case for the structure of p.

```
1  rl   [o]    : < o, PI > => < l, empty > .
2  rl   [MAT]  : < (F = N) · P, PI > => < P, filter(PI,F,N)> .
3  crl  [NEG]  : < (~ Q) · P, PI > => < P, PI \ PI' >
4       if < Q, PI > => < l, PI' > .
5  rl   [ASG]  : < (F ← N) · P, PI > => < P, update(PI,F,N)>  .
6  crl  [COM]  : < (P + Q) · R, PI > => < R, (PI1, PI2) >
7       if < P, PI > => < l, PI1 > /\ < Q, PI > => < l, PI2 > .
8  rl   [KLE-0] : < (P *) · R, PI > => < R, PI >
9  rl   [KLE-1] : < (P *) · R, PI > => < P · R, PI1 > .
10 rl   [KLE-n] : < (P *) · R, PI > => < P · (P *) · R, PI > .
```

The first rule specifies the semantics of *Drop*, i.e., 0 in NetKAT. The rule in Line 2 formalizes the operational semantics of *match*. In the rule, `F`, `N`, `P` and `PI` are Maude variables of sort `Field`, `Nat`, `Policy` and `PackSet`. These variables are universally quantified. Thus, `(F = N)` represents an arbitrary match, `P` an arbitrary policy, and `PI` an arbitrary set of packets. After the execution of `(F = N)`, those packets whose value of the field `F` is not `N` are removed from `PI`. The rule in Line 3 formalizes the case of negation. It is worth mentioning that the condition in the rule is a *rewrite condition*, meaning that the rule takes place if there is a transition from `< Q, PI >` to `< l, PI' >`. The transition means that after executing `Q` on a set `PI` of packets, we obtain a new set `PI'` of packets. According to the operational semantics of negation, the packets that are in both `PI'` and `PI` must be removed from `PI` after `~ Q` is executed, as defined by the body of the rule. The last three rules formalize the operational semantics of Kleene star for the cases of executing policy `P` by zero, one or more times, respectively.

5 Automatic Reasoning for NetKAT

By the executable operational semantics we can perform various formal analysis on NetKAT policies using Maude's built-in functionalities such as simulation, state space exploration and LTL model checking.

5.1 Reachability analysis by state exploration

Given a policy `p` and a set `PI` of packets, one fundamental analysis is to verify if the packets in `PI` will eventually reach their destination. Because all the rules except for those about Kleene star are deterministic, there is one and only one result if in `p` there is no Kleene star. We can check the reachability problem by calculating the execution result using Maude's `rewrite` command, i.e., `rew <p, PI>`, which *simulates* the execution of the policy on `PI` using the rewrite rules defined for the operational semantics of NetKAT.

If there are Kleene stars in `p`, the results after applying `p` on `PI` may be multiple. If that is the case, it is important to verify if some desired result can be obtained by applying `p` to `PI`. It is equivalent to checking the reachability from the initial configuration `< p, PI >` to some desired destination `< p', PI' >`, where `p'` is the remaining policy to execute when `PI'` is reached after applying `p` to `PI`. The reachability verification can be achieved by Maude's state exploration function using `search` command as follows:

```
search [m,n] < p, PI > =>* < p', PI' > [such that condition] .
```

In the square brackets are optional arguments of the command, where m and n are natural numbers specifying the expected number of solutions and the maximal rewriting steps, and *condition* is a Boolean term specifying the condition that target configurations must satisfy.

5.2 Model checking of LTL properties of NetKAT

Using Maude LTL model checker, we can verify not only the reachability of packets with respect to a policy, but also some temporal properties that the policy needs to satisfy. Temporal properties of a policy are used to describe the behavior that the policy should have when packets are transmitted in the network. By model checking the temporal properties, we study the process of packet transmission as well as the transmission result.

The usage of Maude LTL model checker follows the conventional methodology for model checking, i.e., we need first define state propositions, then define LTL formulas with the state propositions and logical as well as temporal connectors, and finally do model checking with a fixed initial state and an LTL formula.

```
mod NETKAT-LTL-MODELCHECKING is
  including OPERATIONAL-SEMANTICS + MODEL-CHECKER .
  subsort Configuration < State .
  ops hasPS hasDS : Int Int -> Prop . vars SW PR DS : Int .
  var P  : Policy . var PS : PackSet . var PK : Packet .
  eq < P , PS > |= hasDS(SW,PR) = checkHasDS(PS,DS,PR) .
  eq < P , PS > |= hasPS(SW,PR) = checkHasPS(PS,SW,PR) .
  eq C:Configuration |= PP:Prop = false [owise] .
endm
```

As an example, we show model checking of forwarding traces of packets in Maude. In the above Maude module two state propositions `hasDS` and `hasPS` are defined.

Given a packet PR, a switch DS and a configuration <p,PS>, hasDS returns true if there is a packet PR in PS whose destination is DS. The other one i.e., hasPS, returns true if there is a packet PR at SW in PS. They are defined by two equations at Lines 6 and 7, where two auxiliary predicates are needed. We omit the detailed definition of the two predicates due to space limitation.

With predefined state propositions we can define and model check LTL properties that are composed by the propositions and LTL operators. For instance, the first command below is used to model check whether a packet X whose destination is switch Y eventually reaches the switch Y with respect to policy p.

```
red modelCheck(< p, PI >, [](hasDS(X,Y) -> <> hasPS(X,Y))) .
red modelCheck(< p, PI >, [](hasPS(X,Y1)/\hasDS(X,Y2) -> <>
    hasPS(X,Y2))) .
```

The second command above is used to verify the property that wherever a packet X is, e.g., Y1, it must be eventually delivered to switch Y2 if its destination is Y2. This verification is more general than the first one in that we can verify the reachability of two arbitrary switches in a topology w.r.t. to a forwarding policy.

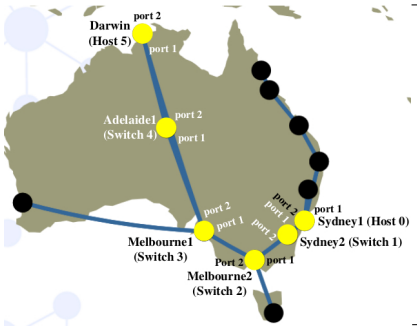
5.3 Equivalence proving by normalization

By Theorem 5 we can verify the equivalence of two policies p, q by reducing them to their normal forms and checking if they are syntactically equal. To automate the process, we formalize the normalization of policies based on the proof of Lemma 4 in Maude.

We declare a function norm which takes two arguments, i.e., a policy in NetKAT and a set of field information, and returns the normal form of the policy. Part of the declaration and the definition of norm is listed as follows:

```
op  norm : Policy FieldRangeSet -> NormalForm .
ceq norm(F = N, FS) = (if PS =/= empty then normPred(F = N
    · PS) else normPred(F = N) fi) if PS := com(rm(FS,F)) .
eq  norm(~(F = N), FS) = normPred((~(F = N)) · com(FS)) .
eq  norm(F ← N, FS) = normPoli(com(FS),(F ← N)) .
eq  norm(PL1 + PL2, FS) = norm(PL1, FS) + norm(PL2, FS) .
eq  norm(PD · PD1, FS) = product(norm(PD, FS),norm(PD1,FS)).
...
ceq norm((PL) *, FS) = normPred(com(FS)) + NF
    if NF := norm(PL,FS) /\ uniform(NF) .
ceq norm((PL) *, FS) = product(NF2, normPred(com(FS)) + NF3)
    if NF := norm(PL,FS) /\ not uniform(NF) /\
       (AT,PL1)+NF1 := NF /\ NF2 := norm(nf2pol(NF1)*,FS) /\
       NF3 := product((AT,PL1),NF2).
```

We take the formalization of the normalization of match and Kleene star for examples. The equation in Line 2 formalizes the normalization of match F = N with respect to a set FS of field information. In the equation, com is a function which takes a set of field information such as $\{(f_1, m_1), (f_2, m_2), \ldots, (f_k, m_k)\}$



(a) The graphical network topology

$$
\begin{array}{rcl}
p & \triangleq & (\texttt{dst} = 0) \cdot (\texttt{pt} \leftarrow 1) + (\texttt{dst} = 5) \cdot (\texttt{pt} \leftarrow 2) \\
t & \triangleq & ((\texttt{sw} = 0) \cdot (\texttt{sw} \leftarrow 1) \cdot (\texttt{pt} \leftarrow 1))+ \\
 & & ((\texttt{sw} = 1) \cdot (\texttt{pt} = 1) \cdot (\texttt{sw} \leftarrow 0))+ \\
 & & ((\texttt{sw} = 1) \cdot (\texttt{pt} = 2) \cdot (\texttt{sw} \leftarrow 2) \cdot (\texttt{pt} \leftarrow 1))+ \\
 & & ((\texttt{sw} = 2) \cdot (\texttt{pt} = 1) \cdot (\texttt{sw} \leftarrow 1) \cdot (\texttt{pt} \leftarrow 2))+ \\
 & & ((\texttt{sw} = 2) \cdot (\texttt{pt} = 2) \cdot (\texttt{sw} \leftarrow 3) \cdot (\texttt{pt} \leftarrow 1))+ \\
 & & ((\texttt{sw} = 3) \cdot (\texttt{pt} = 1) \cdot (\texttt{sw} \leftarrow 2) \cdot (\texttt{pt} \leftarrow 2))+ \\
 & & ((\texttt{sw} = 3) \cdot (\texttt{pt} = 2) \cdot (\texttt{sw} \leftarrow 4) \cdot (\texttt{pt} \leftarrow 1))+ \\
 & & ((\texttt{sw} = 4) \cdot (\texttt{pt} = 1) \cdot (\texttt{sw} \leftarrow 3) \cdot (\texttt{pt} \leftarrow 2))+ \\
 & & ((\texttt{sw} = 4) \cdot (\texttt{pt} = 2) \cdot (\texttt{sw} \leftarrow 5))+ \\
 & & ((\texttt{sw} = 5) \cdot (\texttt{sw} \leftarrow 4) \cdot (\texttt{pt} \leftarrow 2))
\end{array}
$$

(b) Forwading policy and topology in NetKAT

Fig. 1: An example of Australia Network named Aarnet

with each $m_i \in \mathbb{N}$ $(1 \leq i \leq k)$, and returns a predicate

$$\sum_{x_1 \leq m_1, x_2 \leq m_2, \ldots, x_n \leq m_n} (f_1 = x_1) \cdot (f_2 = x_2) \cdot \ldots \cdot (f_k = x_k). \tag{6}$$

Each summand of the predicate and the match forms an atom α. The customized function `normPred` returns a parallel composition of all $\alpha \cdot \sigma_\alpha$.

The last two equations define the normalization of the Kleene star of a policy, e.g., `PL *`, where `PL` is a policy. The equation in Line 9 defines the case where the normal form of `PL` is uniform. The last equation recursively defines the non-uniform case. If the normal form `NF` of the policy `PL` is not uniform, it can be rewritten in the form of `(AT, PL1) + NF1` where `AT` is an atom, `PL1` is the complete assignment of `AT`, and `(AT, PL1)` is a Maude representation of `AT` $\cdot$ `PL1`. We then compute the normal form of `NF1` and denote it by `NF2`. Using `KAT-Denesting` we obtain the normal form of `PL`, as defined by the right-hand term in the body of the equation. The equation formalizes the normalization of Kleene star when the summands in `PL` are not uniform, as described in the proof of Lemma 4.

6 Experiments and Evaluation

In this section, we evaluate the proposed approach by formally verifying the reachability of nodes in the network topologies defined in the website named Internet Topology Zoo [1].

As a concrete example, we consider the network topology highlighting the connection between Sydney and Darwin which is depicted in Figure 1(a). There is a path between Sydney1 and Darwin. It is marked by yellow nodes in the network. The path can be formalized as t that specifies all the bi-directional links along the path. We declare a forwarding policy p for the switches in that path, which are defined in NetKAT as shown in Figure 1(b).

The following `search` command verifies the reachability between Sydney1 and Darwin:

```
search [1] < (p·t)*, | (dst : 5) (sw : 0) (pt : X:Nat) > =>!
           < l | (dst : 5) (sw : 5) (pt : Y:Nat) > .
```

Maude returns one solution with `Y` being instantiated to be 2. It means that there indeed exists a path, along which packets can reach node 5 from port 2 of node 0 after applying the policy $(p \cdot t)^*$.

Searching only shows the reachability of two nodes but cannot guarantee a packet sent from node 0 must eventually reach node 5 based on the result. Such property can be verified by Maude LTL model checking as explained in Section 5.2. The following command is used to verify the property.

```
red modelCheck(< (p·t)*, (dst : 5)(sw : 0)(pk : 1) >,
    [](hasPS(1,0) /\ hasDS(1,5) -> <> hasPS(1,5))) .
```

Maude returns true with the above command, which means that if there is a packet in Host 0 (Syndey1) with destination being Host 5 (Darwin), the packet must eventually reach Darwin.

Another alternative of verifying the reachability between two nodes in a network is to prove the equivalence of a specification and an implementation by normalization, as explained in Section 5.3. In this example, we define a specification policy s which only specifies the effect, but ignores the concrete implementation. For instance, the first summand specifies that all the packets sent from node 0 to node 5 must arrive node 5. The policy i formalizes the implementation of sending/receiving packets along the path described by t between nodes 0 and 5.

$$s \triangleq ((\mathtt{sw} = 0) \cdot (\mathtt{dst} = 5) \cdot (\mathtt{sw} \leftarrow 5) \cdot (\mathtt{pt} \leftarrow 2)) + ((\mathtt{sw} = 5) \cdot (\mathtt{dst} = 0) \cdot (\mathtt{sw} \leftarrow 0) \cdot (\mathtt{pt} \leftarrow 1)) \quad \text{(Specification)}$$

$$i \triangleq ((\mathtt{sw} = 0) \cdot (\mathtt{dst} = 5)) \cdot ((p \cdot t)^*) \cdot ((\mathtt{sw} = 5) \cdot (\mathtt{pt} = 2)) + ((\mathtt{sw} = 5) \cdot (\mathtt{dst} = 0)) \cdot ((p \cdot t)^*) \cdot ((\mathtt{sw} = 0) \cdot (\mathtt{pt} = 1)) \quad \text{(Implementation)}$$

We prove that the specification and the implementation are equal by checking their normal forms are the same with the following command:

```
red norm(i,(sw,5)(pt,2)(dst,5))==norm(s,(sw,5)(pt,2)(dst,5)).
```

Maude returns true, meaning that the specification and the implementation are operationally equivalent. Therefore, packets can be routed from node 0 to node 5, and vice versa.

We verify the reachability property of eight network topologies in Internet Topology Zoo. For each we use three different approaches i.e., searching, model checking, and normalization, as we explained above. Table 4 shows the verification results. All the experiments are conducted on a desktop running Ubuntu 15.10 with an Intel(R) Core(TM) i5-4590 @ 3.30GHz CPU and 2.00GB memory. The data shows that as far as reachability properties are concerned, it is faster to search a desired path by executing the operational semantics than to check the equivalence of two policies by normalization. The inefficiency of normalization

Table 4: Reachability verification of the network topologies in Internet Topology Zoo using searching, model checking and normalization

Network name	Nodes	By searching		By model checking		By normalization	
		Result	Time	Result	Time	Result	Time
Aarnet	3	✓	1ms	✓	0ms	✓	2.12m
Bellsouth	4	✓	0ms	✓	1ms	✓	9.53m
Bellsouth	5	✓	0ms	✓	1ms	✓	38.47m
Aarnet	6	✓	4ms	✓	2ms	✓	1.92h
Aarnet	7	✓	6ms	✓	3ms	✓	5.84h
Aarnet	8	✓	5ms	✓	3ms	✓	10.52h
Aarnet	9	✓	8ms	✓	3ms	✓	21.89h
Aarnet	10	✓	8ms	✓	3ms	✓	23.13h

can be explained as follows. As we can see from (6), to obtain the normal form of a policy we need to do the Cartesian product of terms. When the number of nodes in a network increases, the size of the normal form of the term that describes the network topology will grow exponentially. Maude expands terms according to our definition of normal form without any optimization, which makes normalization a very time-consuming process. However, equivalence checking can be used for verifying other network properties such as loop-freedom and translation validation [8]. It also shows that model checking has a better performance than searching with the increment of node numbers.

7 Concluding Remarks

We have proposed an operational semantics for NetKAT and shown that it is sound and complete with respect to the axiomatic semantics given by Anderson et al. in their seminal paper. We have also formalized the operational semantics and the equational theory of NetKAT in Maude, which allows us to normalize NetKAT expressions and to check if two expressions are equivalent. In addition, we have investigated other verification techniques including searching and model checking. They constitute a formal approach of reasoning about NetKAT expressions with applications such as checking reachability properties in networks. The full Maude code is available online [2]. To our knowledge, the current work is the first to employ a rewrite engine for manipulating NetKAT expressions so as to verify network properties.

As mentioned in Section 1, NetKAT is proprosed in [3], with its axiomatic and denotational semantics carefully defined by building upon previous work on Kleene algebra and earlier network programming languages; see the references in the aforementioned work. In order to verify network properties, it is crucial to develop highly efficient algorithms for checking the equivalence of NetKAT expressions. An attempt in this direction is the coalgebraic decision procedure proposed in [8]. It first converts two NetKAT expressions into two automata by

Brzozowski derivatives, and then tests if the auttomata are bisimilar. On the other hand, our approach heavily relies on the rewriting of NetKAT expressions into normal forms. In terms of time efficiency, unfortunately, both the coalgebraic decision procedure in [8] and our rewriting-based approach are not satisfactory when handling large networks. Therefore, an interesting future work is to pursue faster algorithms for checking the equivalence of NetKAT expressions.

References

1. The Internet Topology Zoo Website. http://www.topology-zoo.org.
2. The Maude Code. https://github.com/zhmtechie/NetKAT-Maude.
3. C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL 2014*, pages 113–126. ACM, 2014.
4. R. Beckett, M. Greenberg, and D. Walker. Temporal NetKAT. In *Proc. PLDI 2016*, pages 386–401. ACM, 2016.
5. M. Clavel, F. Durán, S. Eker, et al., editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
6. N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *Proc. ICFP 2011*, pages 279–291. ACM, 2011.
7. N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva. Probabilistic NetKAT. In *Proc. ESOP 2016*, volume 9632 of *LNCS*, pages 282–309. Springer, 2016.
8. N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for NetKAT. In *Proc. POPL 2015*, pages 343–355. ACM, 2015.
9. A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. In *Proc. PLDI 2013*, pages 483–494. ACM, 2013.
10. D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19(3):427–443, 1997.
11. C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *Proc. POPL 2012*, pages 217–230. ACM, 2012.
12. C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software defined networks. In *Proc. NSDI 2013*, pages 1–13. USENIX Association, 2013.
13. A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *J. Log. Algebr. Program.*, 67(1-2):226–293, 2006.
14. A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *Proc. PADL 2011*, volume 6539 of *LNCS*, pages 235–249. Springer, 2011.
15. A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: simplifying SDN programming using algorithmic policies. In *Proc. SIGCOMM 2013*, pages 87–98. ACM, 2013.