

DyNetKAT: An Algebra of Dynamic Networks^{*}

Georgiana Caltais¹[0000–0002–8653–2299], Hossein Hojjat²[0000–0002–4743–8750],
Mohammad Reza Mousavi³[0000–0002–4869–6794], and
Hünkar Can Tunç^{1,4}[0000–0001–9125–8506]

¹ University of Konstanz, Germany
`georgiana.caltais@uni-konstanz.de`

² TeIAS, Khatam University & University of Tehran, Iran
`hojjat@ut.ac.ir`

³ King’s College London, UK
`mohammad.mousavi@kcl.ac.uk`

⁴ Aarhus University, Denmark
`tunc@cs.au.dk`

Abstract. We introduce a formal language for specifying dynamic updates for Software Defined Networks. Our language builds upon Network Kleene Algebra with Tests (NetKAT) and adds constructs for synchronisations and multi-packet behaviour to capture the interaction between the control- and data-plane in dynamic updates. We provide a sound and ground-complete axiomatisation of our language. We exploit the equational theory to provide an efficient reasoning method about safety properties for dynamic networks. We implement our equational theory in DyNetiKAT – a tool prototype, based on the Maude Rewriting Logic and the NetKAT tool, and apply it to a case study. We show that we can analyse the case study for networks with hundreds of switches using our tool prototype.

Keywords: Software Defined Networks · Dynamic Updates · Dynamic Network Reconfiguration · NetKAT · Process Algebra · Equational Reasoning.

1 Introduction

There is a spectrum of mathematically inspired network programming languages that varies between those with a small number of language constructs and those with expressive language design which allow them to support more networking features. Flowlog [15] and Kinetic [11] are points on the more expressive side of the spectrum, which provide support for formal reasoning based on SAT-solving and model checking, respectively. NetKAT [2, 9] is an example of a

^{*} The work of Georgiana Caltais and Hünkar Can Tunç was supported by the DFG project “CRENKAT”, proj. no. 398056821. The work of Mohammad Reza Mousavi was supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/1. The authors would like to thank Alexandra Silva and Tobias Kappé for their useful insight into the NetKAT framework.

minimalist language based on Kleene algebra with tests that has a sound and complete equational theory. While the core of the language is very simple with a few number of operators, the language has been extended in various ways to support different aspects of networking such as congestion control [8], history-based routing [5] and higher-order functions [19].

Our starting point is NetKAT, because it provides a clean and analysable framework for specifying SDNs. The minimalist design of NetKAT does not cater for some common (failure) patterns in SDNs, particularly those arising from dynamic reconfiguration and the interaction between the data- and control-plane flows. In [12], the authors have proposed an extension to NetKAT to support stateful network updates. The extension embraces the notion of mutable state in the language which is in contrast to its pure functional nature. The purpose of this paper is to propose an extension of NetKAT to support dynamic and stateful behaviours. On the one hand, we preserve the big-step denotational semantics of NetKAT-specific constructs enabling, for instance, handling flow table updates atomically, in the spirit of [16]. On the other hand, we extend NetKAT in a modular fashion, to integrate concurrent SDN behaviours such as dynamic updates, defined via a small-step operational semantics. To this end, we pledge to keep the minimalist design of NetKAT by adding only a few new operators. Furthermore, our extension does not contradict the nature of the language.

A number of concurrent extensions of NetKAT have been introduced to date [10, 17, 20]. These extensions followed different design decisions than the present paper and a comparison of their approaches with ours is provided in Section 2; however, the most important difference lies in the fact that inspired by earlier abstractions in this domain [16], we were committed to create different layers for data-plane flows and dynamic updates such that every data-plane packet observes a single set of flow tables through its flight through the network. This allowed us, unlike the earlier approaches, to build a layer on top of NetKAT without modifying its semantics. Although our presentation in this paper is based on NetKAT, we envisage that our concurrency layer can be modularly (in the sense of Modular SOS [13]) used for other network programming languages in the above-mentioned spectrum. We leave a more careful investigation of the modularity on other network languages for future work.

Running Example. To illustrate our language concepts, we focus on modelling with DyNetKAT an example of a stateful firewall that involves dynamically updating the flow table. The example is overly simplified for the purpose of presentation. Towards the end of this paper and also in the extended version [6], we treat more complex and larger-scale case studies to evaluate the applicability and analysability of our language.

Example 1. A firewall is supposed to protect the intranet of an organisation from unauthorised access from the Internet. However, due to certain requests from the intranet, it should be able to open up connections from the Internet to intranet. An example is when a user within the intranet requests a secure connection to a node on the Internet; in that case, the response from the node should be allowed

to enter the intranet. The behaviour of updating the flow tables with respect to some events in the network such as receiving a specific packet is a challenging phenomenon for languages such as NetKAT.

Figure 1 shows a simplified version of the stateful firewall network. Note that we are not interested in the flow of packets but interested in the flow update. In this version, the *Switch* does not allow any packet from the port *ext* to *int* at the beginning. When the *Host* sends a request to the *Switch* it opens up the connection.

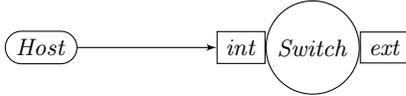


Fig. 1: Stateful Firewall

Our Contributions. The contributions of this paper are summarised as follows:

- we define the syntax and operational semantics of a dynamic extension of NetKAT that allows for modelling and reasoning about control-plane updates and their interaction with data-plane flows (Sections 2.3, 2.4);
- we give a sound and ground-complete axiomatisation of our language (Section 3); and
- we devise analysis methods for reasoning about flow properties using our axiomatisation, apply them on examples from the domain and gather and analyse evidence of applicability and efficiency for our approach (Sections 4, 5).

Structure of Paper. In Section 2, we provide a brief overview of NetKAT, review our design decisions and introduce the syntax and operational semantics of DyNetKAT. In Section 3, we investigate some semantic properties of DyNetKAT by defining a notion of behavioural equivalence and providing a sound and ground-complete axiomatisation. We exploit this axiomatisation in Section 4 in an analysis method. We implement and apply our analysis method in Section 5 on a case study and report about its scalability on large examples with hundreds of switches. We conclude the paper and present some avenues for future work in Section 6.

2 Language Design

In what follows, we provide a brief overview of the NetKAT syntax and semantics [2]. Then, we motivate our language design decisions, we introduce the syntax of DyNetKAT and its underlying semantics, and provide the corresponding encoding of our running examples presented in Section 1.

2.1 Brief Overview of NetKAT

We proceed by first introducing some basic notions that are used throughout the paper.

Definition 1 (Network Packets.) Let $F = \{f_1, \dots, f_n\}$ be a set of field names f_i with $i \in \{1, \dots, n\}$. We call network packet a function in $F \rightarrow \mathbb{N}$ that maps field names in F to values in \mathbb{N} . We use σ, σ' to range over network packets. We write, for instance, $\sigma(f_i) = v_i$ to denote a test checking whether the value of f_i in σ is v_i . Furthermore, we write $\sigma[f_i := n_i]$ to denote the assignment of f_i to v_i in σ .

A (possibly empty) list of packets is formally defined as a function from natural numbers to packets, where the natural number in the domain denotes the position of the packet in the list such that the domain of the function forms an interval starting from 0.

The empty list is denoted by $\langle \rangle$ and is formally defined as the empty function (the function with the empty set as its domain). Let σ be a packet and l be a list, then $\sigma :: l$ is the list l' in which σ is at position 0 in l' , i.e., $l'(0) = \sigma$, and $l'(i+1) = l(i)$, for all i in the domain of l .

In Figure 2, we recall the NetKAT syntax and semantics [2].

NetKAT Syntax:

$$\begin{aligned} Pr &::= \mathbf{0} \mid \mathbf{1} \mid Pr + Pr \mid Pr \cdot Pr \mid \neg Pr \\ N &::= Pr \mid f \leftarrow n \mid N + N \mid N \cdot N \mid N^* \mid \mathbf{dup} \end{aligned}$$

NetKAT Semantics:

$$\begin{aligned} \llbracket \mathbf{1} \rrbracket (h) &\triangleq \{h\} \\ \llbracket \mathbf{0} \rrbracket (h) &\triangleq \{\} \\ \llbracket f = n \rrbracket (\sigma :: h) &\triangleq \begin{cases} \{\sigma :: h\} & \text{if } \sigma(f) = n \\ \{\} & \text{otherwise} \end{cases} \\ \llbracket \neg a \rrbracket (h) &\triangleq \{h\} \setminus \llbracket a \rrbracket (h) \\ \llbracket f \leftarrow n \rrbracket (\sigma :: h) &\triangleq \{\sigma[f := n] :: h\} \\ \llbracket p + q \rrbracket (h) &\triangleq \llbracket p \rrbracket (h) \cup \llbracket q \rrbracket (h) \\ \llbracket p \cdot q \rrbracket (h) &\triangleq (\llbracket p \rrbracket \bullet \llbracket q \rrbracket) (h) \\ \llbracket p^* \rrbracket (h) &\triangleq \bigcup_{i \in \mathbb{N}} F^i (h) \\ F^0 (h) &\triangleq \{h\} \\ F^{i+1} (h) &\triangleq (\llbracket p \rrbracket \bullet F^i) (h) \\ (f \bullet g)(x) &\triangleq \bigcup \{g(y) \mid y \in f(x)\} \\ \llbracket \mathbf{dup} \rrbracket (\sigma :: h) &\triangleq \{\sigma :: (\sigma :: h)\} \end{aligned}$$

Fig. 2: NetKAT: Syntax and Semantics [2]

The predicate for dropping a packet is denoted by $\mathbf{0}$, while passing on a packet (without any modification) is denoted by $\mathbf{1}$. The predicate checking whether the field f of a packet has value n is denoted by $(f = n)$; if the predicate fails

on the current packet it results on dropping the packet, otherwise it will pass the packet on. Disjunction and conjunction between predicates are denoted by $Pr + Pr$ and $Pr \cdot Pr$, respectively. Negation is denoted by $\neg Pr$. Predicates are the basic building blocks of NetKAT policies and hence, a predicate is a policy by definition. The policy that modifies the field f of the current packet to take value n is denoted by $(f \leftarrow n)$. A multicast behaviour of policies is denoted by $N + N$, while sequencing policies (to be applied on the same packet) are denoted by $N \cdot N$. The repeated application of a policy is encoded as N^* . The construct **dup** simply makes a copy of the current network packet.

In [2], lists of packets are referred to as *histories*. Let H stand for the set of packet histories, and $\mathcal{P}(H)$ denote the powerset of H . More formally, the denotational semantics of NetKAT policies is inductively defined via the semantic map $\llbracket - \rrbracket : N \rightarrow (H \rightarrow \mathcal{P}(H))$ in Figure 2, where N stands for the set of NetKAT policies, $h \in H$ is a packet history, $a \in Pr$ denotes a NetKAT predicate and $\sigma \in F \rightarrow \mathbb{N}$ is a network packet.

For a reminder, the equational axioms of NetKAT include the Kleene Algebra axioms, Boolean Algebra axioms and the so-called Packet Algebra axioms that handle NetKAT networking specific constructs such as field assignments and **dup**. In this paper, we write E_{NK} to denote the NetKAT axiomatisation [2].

2.2 Design Decisions

Our main motivation behind DyNetKAT is to have a *minimalist* language that can model *control-plane* and *data-plane* network traffic and their interaction. Our choice for a minimal language is motivated by our desire to use our language as a basis for scalable analysis. We would like to be able to compile major practical languages into ours. Our minimal design helps us reuse much of the well-known scalable analysis techniques. Regarding its modelling capabilities, we are interested in modelling the stateful and dynamic behaviour of networks emerging from these interactions. We would like to be able to model control messages, connections between controllers and switches, data packets, links among switches, and model and analyse their interaction in a seamless manner.

Based on these motivations, we start off with NetKAT as a fundamental and minimal network programming language, which allows us to model the basic policies governing the network traffic. The choice of NetKAT, in addition to its minimalist nature, is motivated by its rigorous semantics and equational theory, and the existing techniques and tools for its analysis. This motivates our next design constraint, namely, to build upon NetKAT in a hierarchical manner and without redefining its semantics. This constraint should not be taken lightly as the challenges in the recent concurrent extensions of NetKAT demonstrated [10, 17, 20]. We will elaborate on this point, in the presentation of our syntax and semantics. We can achieve this thanks to the abstractions introduced in the domain [16] that allow for a neat layering of data-plane and control-plan flows such that every data-plane flow sees one set of flow-tables in its flight through the network.

We then introduce a few extensions and modifications to cater for the phenomena we desire to model in our extension regarding control-plane and dynamic and stateful behaviour:

- Synchronisation: we introduce a basic mechanism of handshake synchronisation with the possibility of communicating a network program (a flow table). This construct allows for capturing the dynamicity and interaction between the control and data planes.
- Guarded recursion: we introduce the concept of recursion to model the (persistent) dynamic changes that result from control messages and stateful behaviour. In other words, recursion is used to model the new state of the flow tables. An alternative modelling construct could have been using “global” variables and guards, but we prefer recursion due to its neat algebraic representation. We restrict the use of recursion to guarded recursion, that is a policy should be applied before changing state to a new recursive definition, in order to remain within a decidable and analyse-able realm. A natural extension of our framework could introduce formal parameters and parameterised recursive variables; this future extension is orthogonal to our existing extensions and in this paper, we go for a minimal extension in which the parameters are coded in variable names.
- Multi-packet semantics: we introduce the semantics of treating a list of packets, which is essential for studying the interaction between control- and data plane packets. This is in contrast with NetKAT where a single-packet semantics is introduced. The introduction of multi-packet semantics also called for a new operator to denote the end of applying a flow-table to the current packet and proceeding with the next packet (possibly with the modified flow-table in place). This is our new sequential composition operator, denoted by “;”.

2.3 DyNetKAT Syntax

As already mentioned, NetKAT provides the possibility of recording the individual “hops” that packets take as they go through the network by using the so-called **dup** construct. The latter keeps track of the state of the packet at each intermediate hop. As a brief reminder of the approach in [2]: assume a NetKAT switch policy p and a topology t , together with an ingress in and an egress out . Checking whether out is reachable from in reduces to checking: $in \cdot \mathbf{dup} \cdot (p \cdot t \cdot \mathbf{dup})^* \cdot out \neq \mathbf{0}$ (see Definition 2 and Theorem 4 in [2]). Furthermore, as shown in [9], **dup** plays a crucial role in devising the NetKAT language semantics in a coalgebraic fashion, via Brzozowski-like derivatives on top of NetKAT coalgebras (or NetKAT automata) corresponding to NetKAT expressions.

We decided to depart from NetKAT in this respect, due to our important constraint not to redefine the NetKAT semantics: the **dup** expression allows for observable intermediate steps that result from incomplete application of flow-tables and in concurrency scenarios, the same data packet may become subject to more than one flow table due to the concurrent interactions with the control

plane. For this semantics to be compositional, one needs to define a small step operational semantics in such a way that the small steps in predicate evaluation also become visible (see our past work on compositionality of SOS with data on such constraints [14]). This will first break our constraint in building upon NetKAT semantics and secondly, due to the huge number of possible interleavings, make the resulting state-space intractable for analysis.

In addition to the argumentation above, note that similarly to the approach in [2], we work with packet fields ranging over finite domains. Consequently, our analyses can be formulated in terms of reachability properties, further verifiable by means of **dup**-free expressions of shape: $in \cdot (p \cdot t)^* \cdot out \neq \mathbf{0}$. Hence, we chose to define DyNetKAT synchronisation, guarded recursion and multi-packet semantics on top of the **dup**-free fragment of NetKAT, denoted by $\text{NetKAT}^{-\text{dup}}$.

The syntax of DyNetKAT is defined on top of the **dup**-free fragment of NetKAT as:

$$\begin{aligned} N &::= \text{NetKAT}^{-\text{dup}} \\ D &::= \perp \mid N ; D \mid x?N ; D \mid x!N ; D \mid D \parallel D \mid D \oplus D \mid X \\ X &\triangleq D \end{aligned} \quad (1)$$

We write $p \in \text{NetKAT}$, $p \in \text{NetKAT}^{-\text{dup}}$ or, respectively, $p \in \text{DyNetKAT}$ in order to refer to a NetKAT, $\text{NetKAT}^{-\text{dup}}$ or, respectively, DyNetKAT policy p .

The DyNetKAT-specific constructs are as follows. By \perp we denote a dummy policy without behaviour. Our new sequential composition operator, denoted by $N ; D$, specifies when the $\text{NetKAT}^{-\text{dup}}$ policy N applicable to the current packet has come to a successful end and, thus, the packet can be transmitted further and the next packet can be fetched for processing according to the rest of the policy D .

Communication in DyNetKAT, encoded via $x!N ; D$ and $x?N ; D$, consists of two steps. In the first place, sending and receiving $\text{NetKAT}^{-\text{dup}}$ policies through channel x are denoted by $x!N$, and $x?N$. Intuitively, these correspond to updating the current network configuration according to N . Secondly, as soon as the sending or receiving messages are successfully communicated, a new packet is fetched and processed according to D . The parallel composition of two DyNetKAT policies (to enable synchronisation) is denoted by $D \parallel D$.

As it will become clearer in Section 2.4, communication in DyNetKAT guarantees preservation of well-defined behaviours when transitioning between network configurations. This corresponds to the so-called per-packet consistency in [16], and it guarantees that every packet traversing the network is processed according to exactly one $\text{NetKAT}^{-\text{dup}}$ policy.

Non-deterministic choice of DyNetKAT policies is denoted by $D \oplus D$. For a non-deterministic choice over a finite domain P , we use the syntactic sugar $\bigoplus_{p \in P} P'$, where p appears as “bound variable” in P' ; this is interpreted as a sum of finite summand by replacing the variable p with all its possible values in P .

Finally, one can use recursive variables X in the specification of DyNetKAT policies, where each recursive variable should have a unique defining equation $X \triangleq D$.

For the simplicity of notation, we do not explicitly specify the trailing “; \perp ” in our policy specifications, whenever clear from the context.

In Figure 3 we provide the DyNetKAT formalisation of the firewall in Example 1. In the DyNetKAT encoding, we use the message channel *secConReq* to open up the connection and *secConEnd* to close it. We model the behaviour of the switch using the two programs *Switch* and *Switch'*.

$$\begin{aligned} \text{Switch} \triangleq & ((\text{port} = \text{int}) \cdot (\text{port} \leftarrow \text{ext})) ; \text{Switch} \oplus \\ & ((\text{port} = \text{ext}) \cdot \mathbf{0}) ; \text{Switch} \oplus \\ & \text{secConReq?}\mathbf{1} ; \text{Switch}' \end{aligned}$$

$$\begin{aligned} \text{Switch}' \triangleq & ((\text{port} = \text{int}) \cdot (\text{port} \leftarrow \text{ext})) ; \text{Switch}' \oplus \\ & ((\text{port} = \text{ext}) \cdot (\text{port} \leftarrow \text{int})) ; \text{Switch}' \oplus \\ & \text{secConEnd?}\mathbf{1} ; \text{Switch} \end{aligned}$$

$$\text{Host} \triangleq \text{secConReq!}\mathbf{1} ; \text{Host} \oplus \text{secConEnd!}\mathbf{1} ; \text{Host}$$

$$\text{Init} \triangleq \text{Host} \parallel \text{Switch}$$

Fig. 3: Stateful Firewall in DyNetKAT

2.4 DyNetKAT Semantics

The operational semantics of DyNetKAT in Figure 4 is provided over configurations of shape (d, H, H') , where d stands for the current DyNetKAT policy, H is the list of packets to be processed by the network according to d and H' is the list of packets handled successfully by the network. The rule labels γ range over pairs of packets (σ, σ') or communication/reconfiguration-like actions of shape $x!q, x?q$ or $\mathbf{rcfg}(\mathbf{x}, \mathbf{q})$, depending on the context.

Note that the DyNetKAT semantics is devised in a “layered” fashion. Rule $(\mathbf{cpol}'_)$ in Figure 4 is the base rule that makes the transition between the NetKAT denotations and DyNetKAT operations. More precisely, whenever σ' is a packet resulted from the successful evaluation of a NetKAT policy p on σ , a (σ, σ') -labelled step is observed at the level of DyNetKAT. This transition applies whenever the current configuration encapsulates a DyNetKAT policy of shape $p; q$ and a list of packets to be processed starting with σ . The resulting configuration continues with evaluating q on the next packet in the list, while σ' is marked as successfully handled by the network.

The remaining rules in Figure 4 define non-deterministic choice \oplus , synchronisation \parallel and recursion X in the standard fashion. Note that synchronisations leave the packet lists unchanged. Due to space limitation, we omitted the explicit

| | |
|--|---|
| $(\mathbf{cpol}'_{\cdot}) \frac{\sigma' \in \llbracket p \rrbracket(\sigma::\langle \rangle)}{(p; q, \sigma :: H, H') \xrightarrow{(\sigma, \sigma')} (q, H, \sigma' :: H')}$ | $(\mathbf{cpol}_{\mathbf{x}}) \frac{(p, H_0, H_1) \xrightarrow{\gamma} (p', H'_0, H'_1)}{(X, H_0, H_1) \xrightarrow{\gamma} (p', H'_0, H'_1)} X \triangleq p$ |
| $(\mathbf{cpol}_{\oplus}) \frac{(p, H_0, H'_0) \xrightarrow{\gamma} (p', H_1, H'_1)}{(p \oplus q, H_0, H'_0) \xrightarrow{\gamma} (p', H_1, H'_1)}$ | $(\mathbf{cpol}_{\parallel}) \frac{(p, H_0, H'_0) \xrightarrow{\gamma} (p', H_1, H'_1)}{(p \parallel q, H_0, H'_0) \xrightarrow{\gamma} (p' \parallel q, H_1, H'_1)}$ |
| $(\mathbf{cpol}_{\bullet}) \frac{}{(x \bullet p; q, H, H') \xrightarrow{x \bullet p} (q, H, H')} \bullet \in \{?, !\}$ | |
| $(\mathbf{cpol}_{\clubsuit \spadesuit}) \frac{(q, H, H') \xrightarrow{x \clubsuit p} (q', H, H') \quad (s, H, H') \xrightarrow{x \spadesuit p} (s', H, H')}{(q \parallel s, H, H') \xrightarrow{\mathbf{rcfg}(\mathbf{x}, \mathbf{p})} (q' \parallel s', H, H')} \begin{array}{l} \clubsuit =? \quad \spadesuit =! \\ \text{or} \\ \clubsuit =! \quad \spadesuit =? \end{array}$ | |

$$\gamma ::= (\sigma, \sigma') \mid x!q \mid x?q \mid \mathbf{rcfg}(\mathbf{x}, \mathbf{q})$$

Fig. 4: DyNetKAT: Operational Semantics (relevant excerpt)

definitions of the symmetric cases for \oplus and \parallel . The full semantics is provided in [6].

In Figure 5 we depict a labelled transition system (LTS) encoding a possible behaviour of the stateful firewall in Example 1. We assume the list of network packets to be processed consists of a “safe” packet σ_i travelling from *int* to *ext* (i.e., $\sigma_i(\text{port}) = \text{int}$) followed by a potentially “dangerous” packet σ_e travelling from *ext* to *int* (i.e., $\sigma_e(\text{port}) = \text{ext}$). For the simplicity of notation, in Figure 5 we write H for *Host*, S for *Switch*, S' for *Switch'*, SCR for *secConReq* and SCE for *secConEnd*. Note that σ_e can enter the network only if a secure connection request was received. More precisely, the transition labelled (σ_e, σ_i) is preceded by a transition labelled $SCR?1$ or $\mathbf{rcfg}(SCR, \mathbf{1})$:

$$n_2 \xrightarrow{SCR?1, \mathbf{rcfg}(SCR, \mathbf{1})} n_3 \xrightarrow{(\sigma_e, \sigma_i)} n_4.$$

3 Semantic Results

In this section we define bisimilarity of DyNetKAT policies and provide a corresponding sound and ground-complete axiomatization.

Bisimilarity of DyNetKAT terms is defined in the standard fashion:

Definition 2 (Bisimilarity (\sim)) *A symmetric relation R over DyNetKAT policies is a bisimulation whenever for $(p, q) \in R$ the following holds:*

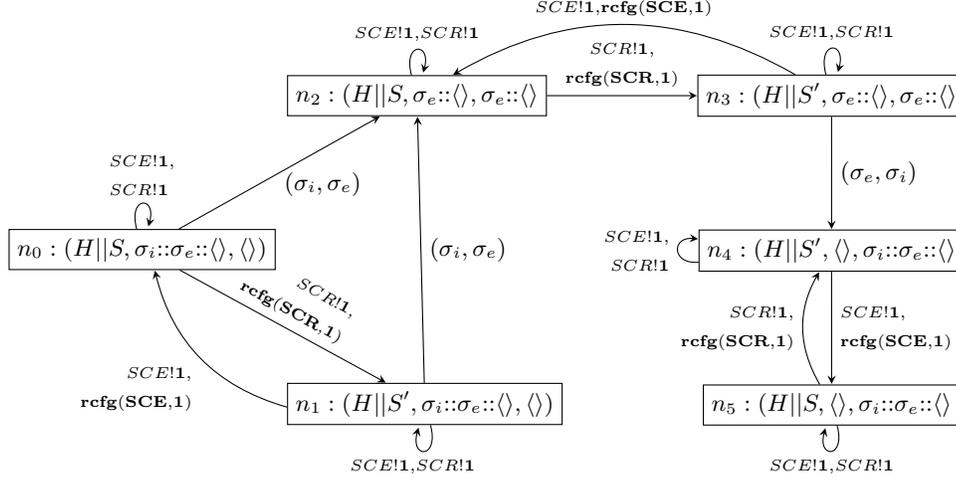


Fig. 5: Stateful Firewall LTS

If $(p, H_0, H_1) \xrightarrow{\gamma} (p', H'_0, H'_1)$ then exists q' s.t. $(q, H_0, H_1) \xrightarrow{\gamma} (q', H'_0, H'_1)$ and $(p', q') \in R$, with $\gamma ::= (\sigma, \sigma') \mid x?r \mid x!r \mid \mathbf{rcfg}(\mathbf{x}, \mathbf{r})$.

We call bisimilarity the largest bisimulation relation. Two policies p and q are bisimilar ($p \sim q$) if and only if there is a bisimulation relation R such that $(p, q) \in R$.

Semantic equivalence of $\text{NetKAT}^{\text{-dup}}$ policies is preserved by DyNetKAT bisimilarity.

Proposition 1 (Semantic Layering). Let p and q be two $\text{NetKAT}^{\text{-dup}}$ policies. The following holds: $\llbracket p \rrbracket = \llbracket q \rrbracket$ iff $(p; d) \sim (q; d)$ for any DyNetKAT policy d .

Proof. This follows according to the definition of bisimilarity and $(\mathbf{cpol}_{\checkmark})$ in Figure 4.

We further provide some additional ingredients needed to introduce the DyNetKAT axiomatisation in Figure 6. First, note that our notion of bisimilarity identifies synchronisation steps as in $(\mathbf{cpol}_{\clubsuit, \spadesuit})$ in Figure 4. At the axiomatisation level, this requires introducing corresponding constants $\mathbf{rcfg}_{x,z}$ defined as:

$$\overline{(\mathbf{rcfg}_{x,z}; p, H_0, H_1) \xrightarrow{\mathbf{rcfg}(x,z)} (p, H_0, H_1)}$$

In accordance with standard approaches to process algebra (see, e.g., [1, 3]) we consider the restriction operator $\delta_{\mathcal{L}}(-)$ with \mathcal{L} a set of forbidden actions ranging over $x?z$ and $x!z$ as in (1). In practice, we use the restriction operator to force synchronous communication. We also define a projection operator

| | |
|--|--|
| for $p, q, r \in \text{DyNetKAT}$ and $z, y \in \text{NetKAT}^{-\text{dup}}$ for $a ::= z \mid x?z \mid x!z \mid \mathbf{rcfg}_{x,z}$ | for $at ::= \alpha \cdot \pi \mid x?z \mid x!z \mid \mathbf{rcfg}_{x,z}$: |
| $\mathbf{0}; p \equiv \perp$ (A0) | $\delta_{\mathcal{L}}(\perp) \equiv \perp$ (δ_{\perp}) |
| $(z + y); p \equiv z; p \oplus y; p$ (A1) | $\delta_{\mathcal{L}}(at; p) \equiv at; \delta_{\mathcal{L}}(p)$ if $at \notin \mathcal{L}$ (δ_{\cdot}) |
| $p \oplus q \equiv q \oplus p$ (A2) | $\delta_{\mathcal{L}}(at; p) \equiv \perp$ if $at \in \mathcal{L}$ (δ_{\cdot}^{\perp}) |
| $(p \oplus q) \oplus r \equiv p \oplus (q \oplus r)$ (A3) | $\delta_{\mathcal{L}}(p \oplus q) \equiv \delta_{\mathcal{L}}(p) \oplus \delta_{\mathcal{L}}(q)$ (δ_{\oplus}) |
| $p \oplus p \equiv p$ (A4) | |
| $p \oplus \perp \equiv p$ (A5) | for $n \in \mathbb{N}$: |
| $p \parallel q \equiv q \parallel p$ (A6) | $\pi_0(p) \equiv \perp$ (Π_0) |
| $p \parallel \perp \equiv p$ (A7) | $\pi_n(\perp) \equiv \perp$ (Π_{\perp}) |
| $p \parallel q \equiv p \parallel q \oplus q \parallel p \oplus p \parallel q$ (A8) | $\pi_{n+1}(at; p) \equiv at; \pi_n(p)$ (Π_{\cdot}) |
| $\perp \parallel p \equiv \perp$ (A9) | $\pi_n(p \oplus q) \equiv \pi_n(p) \oplus \pi_n(q)$ (Π_{\oplus}) |
| $(a; p) \parallel q \equiv a; (p \parallel q)$ (A10) | |
| $(p \oplus q) \parallel r \equiv (p \parallel r) \oplus (q \parallel r)$ (A11) | $p \equiv q$ if $\forall n \in \mathbb{N} : \pi_n(p) \equiv \pi_n(q)$ (AIP) |
| $(x?z; p) \mid (x!z; q) \equiv \mathbf{rcfg}_{x,z}; (p \parallel q)$ (A12) | |
| $(p \oplus q) \mid r \equiv (p \mid r) \oplus (q \mid r)$ (A13) | E_{NK} |
| $p \mid q \equiv q \mid p$ (A14) | |
| $p \mid q \equiv \perp$ [otherwise] (A15) | |

Fig. 6: The axiom system E_{DNK} (including E_{NK})

$\pi_n(-)$ that, intuitively, captures the first n steps of a DyNetKAT policy. $\pi_n(-)$ is crucial for defining the so-called ‘‘Approximation Induction Principle’’ that enables reasoning about equivalence of recursive DyNetKAT specifications. Last, but not least, in our axiomatisation we employ the left-merge operator (\parallel) and the communication-merge operator (\mid) utilised for axiomatising parallel composition. Intuitively, a process of shape $p \parallel q$ behaves like p as a first step, and then continues as the parallel composition between the remaining behaviour of p and q . A process of shape $p \mid q$ forces the synchronous communication between p and q in a first step, and then continues as the parallel composition between the remaining behaviours of p and q . The full description of these auxiliary operators is provided in [6].

From this point onward, we denote by DyNetKAT the extension with the operators $\delta_{\mathcal{L}}(-)$, $\pi_n(-)$ and $\mathbf{rcfg}_{x,z}$:

$$\begin{aligned}
 N &::= \text{NetKAT}^{-\text{dup}} \\
 D_e &::= \perp \mid N; D \mid x?N; D_e \mid x!N; D_e \mid \mathbf{rcfg}_{x,N}; D_e \mid \\
 &D_e \parallel D_e \mid D_e \oplus D_e \mid \delta_{\mathcal{L}}(D_e) \mid \pi_n(D_e) \mid D_e \parallel D_e \mid D_e \mid D_e \mid X \\
 X &\triangleq D_e, n \in \mathbb{N}, \mathcal{L} = \{c \mid c ::= x?N \mid x!N\}
 \end{aligned} \tag{2}$$

Bisimilarity is defined for DyNetKAT terms as in (2) in the natural fashion.

Lemma 3 *For DyNetKAT, bisimilarity is a congruence.*

Proof sketch. The result follows from the fact that the semantic rules defined in this paper comply to the congruence formats proposed in [14]; the notion of bisimilarity used in our paper coincides with the notion of stateless bisimilarity in [14] and hence, the lemma follows. \blacksquare

In Figure 6, we introduce E_{DNK} – the axiom system of DyNetKAT, including the NetKAT axiomatisation E_{NK} . Most of the axioms in Figure 6 comply to the standard axioms of parallel and communicating processes [3], where, intuitively, \oplus plays the role of non-deterministic choice, $;$ resembles sequential composition and \perp is a process that deadlocks. An interesting axiom is (A7) : $p \parallel \perp \equiv p$ which, intuitively, states that if one network component fails, then the whole system continues with the behaviour of the remaining components. This is a departure from the approach in [10], where recovery is not possible in case of a component’s failure; i.e., $e \parallel 0 \equiv 0$. Additionally, (A12) “pin-points” a communication step via the newly introduced constants of form $\mathbf{rcfg}_{x,z}$. Axiom (A0) states that if the current packet is dropped as a result of the unsuccessful evaluation of a NetKAT policy, then the continuation is deadlocked. (A1) enables mapping the non-deterministic choice at the level of NetKAT to the setting of DyNetKAT.

The axioms encoding the restriction operator $\delta_{\mathcal{L}}(-)$ and the projection operator $\pi_n(-)$ are defined in the standard fashion, on top of DyNetKAT normal forms later defined in this section. Intuitively, normal forms are defined inductively, as sums of complete tests and complete assignments $\alpha \cdot \pi$, or communication steps $x?q$, $x!q$ and $\mathbf{rcfg}_{x,q}$, followed by arbitrary DyNetKAT policies. Complete tests (typically denoted by α) and complete assignments (typically denoted by π) were originally introduced in [2]. In short: let $F = \{f_1, \dots, f_n\}$ be a set of fields names with values in V_i , for $i \in \{1, \dots, n\}$. We call *complete test* (resp., *complete assignment*) an expression $f_1 = v_1 \cdot \dots \cdot f_n = v_n$ (resp., $f_1 \leftarrow v_1 \cdot \dots \cdot f_n \leftarrow v_n$), with $v_i \in V_i$, for $i \in \{1, \dots, n\}$. Last, but not least, axiom (AIP) corresponds to the so-called “Approximation Induction Principle”, and it provides a mechanism for reasoning about the equivalence of recursive behaviours, up to a certain limit denoted by n .

In what follows, we show that the axiom system E_{DNK} is sound and ground-complete with respect to DyNetKAT bisimilarity.

Lemma 4 (NetKAT^{-dup} Normal Forms) *We call a NetKAT^{-dup} policy q in normal form (n.f.) whenever q is of shape $\Sigma_{\alpha \cdot \pi \in \mathcal{A}} \alpha \cdot \pi$ with $\mathcal{A} = \{\alpha_i \cdot \pi_i \mid i \in I\}$. E_{NK} is normalising for NetKAT^{-dup}.*

Proof sketch. The result follows from Lemma 4 in [2] stating that the standard semantics of every NetKAT expression is equal to the union of its minimal nonzero terms. In the context of NetKAT^{-dup} and packet values drawn from finite domains (as is the case in [2]), this union can be equivalently expressed as a sum of complete tests and complete assignments. I.e., $\vdash r \equiv \Sigma_{i \in I} \alpha_i \cdot \pi_i$ for every NetKAT^{-dup} expression r . ■

Definition 5 (DyNetKAT Normal Forms) *We call a DyNetKAT policy in normal form (n.f.) if it is of shape*

$$\Sigma_{i \in I}^{\oplus} (\alpha_i \cdot \pi_i); d_i \oplus \Sigma_{j \in J}^{\oplus} c_j; d_j (\oplus \perp)$$

where d_i, d_j range over DyNetKAT policies and $c_j ::= x?q \mid x!q \mid \mathbf{rcfg}_{x,q}$ with q denoting terms in NetKAT^{-dup}.

Definition 6 (Guardedness) A DyNetKAT policy p is guarded if and only if all occurrences of all variables X in p are guarded. An occurrence of a variable X in a policy p is guarded if and only if (i) p has a subterm of shape $p';t$ such that either p' is variable-free, or all the occurrences of variables Y in p' are guarded, and X occurs in t , or (ii) if p is of shape $y?X;t$, $y!X;t$ or $\mathbf{rcfg}_{X,t}$.

Note that guarded DyNetKAT policies are finitely branching.

Lemma 7 (DyNetKAT Normalisation) E_{DNK} is normalising for DyNetKAT.

Proof sketch. The proof follows from Lemma 4 and (A1), by structural induction. *Base cases:* $\boxed{p \triangleq \perp}$ trivially holds; $\boxed{p \triangleq q;d}$ with q a NetKAT^{-dup} term holds by Lemma 4 and (A1); $\boxed{p \triangleq c;d}$ with $c ::= x?q \mid x!q \mid \mathbf{rcfg}_{x,q}$ trivially holds. *Induction step, cases:* $\boxed{p \triangleq X}$ - discarded, as p is not guarded; $\boxed{p \triangleq p_1 \oplus p_2}$; $\boxed{p \triangleq p_1 \parallel p_2}$; $\boxed{p \triangleq \pi_n(p')}$; $\boxed{p \triangleq p_1 \mid p_2}$; $\boxed{p \triangleq \delta_{\mathcal{L}}(p')}$ and, eventually, $\boxed{p \triangleq p_1 \parallel p_2}$. All items before follow by the axiom system E_{DNK} and the induction hypothesis, under the assumption that p_1, p_2 and p' are guarded. ■

In what follows, we assume DyNetKAT policies are guarded.

Lemma 8 (Soundness of $E_{\text{DyNetKAT} \setminus \text{AIP}}$) Let $E_{\text{DyNetKAT} \setminus \text{AIP}}$ stand for the axiom system E_{DNK} in Figure 6, without the axiom (AIP). $E_{\text{DyNetKAT} \setminus \text{AIP}}$ is sound for DyNetKAT bisimilarity.

Proof sketch. This is proven in a standard fashion, by case analysis on transitions of shape

$$(p, H_0, H'_0) \xrightarrow{\gamma} (q, H_1, H'_1)$$

with $\gamma ::= (\sigma, \sigma') \mid x?n \mid x!n \mid \mathbf{rcfg}(\mathbf{x}, \mathbf{n})$, according to the semantic rules of the DyNetKAT operators in (2). Take (A0) for instance. The left hand-side $\mathbf{0};p$ can only evolve according to (\mathbf{cpol}'_{\perp}) in Fig. 4 which, in turn, has an empty premise as $\llbracket \mathbf{0} \rrbracket(\sigma :: \langle \rangle) = \{\}$ for all σ . Thus, (\mathbf{cpol}'_{\perp}) does not entail any step for this case. Symmetrically, there is no semantic transition for \perp in Fig. 4. In other words, none of the left/right hand-sides of (A0) displays any behaviour, therefore the axiom is sound. ■

Lemma 9 (Soundness of AIP) The Approx. Induction Principle (AIP) is sound for DyNetKAT bisimilarity.

Proof sketch. The proof is close to the one of Theorem 2.5.8 in [3] and uses the branching finiteness property of guarded DyNetKAT policies. ■

Theorem 1 (Soundness & Completeness). E_{DNK} is sound and ground-complete for DyNetKAT bisimilarity.

Proof. Soundness: if $E_{DNK} \vdash p \equiv q$ then $p \sim q$, follows from Lemma 8 and Lemma 9.

Completeness: if $p \sim q$ then $E_{DNK} \vdash p \equiv q$, is shown as follows. Without loss of generality, assume p and q are in n.f., according to Lemma 7. We want to show that:

$$p \equiv q \oplus p \quad q \equiv p \oplus q \quad (3)$$

which, by ACI of \oplus implies $p \equiv q$. This reduces to showing that every summand of q and vice-versa. We first argue that every summand of p is a summand of q . The reasoning is by structural induction.

Base case.

- $p \triangleq \perp$. It holds by the hypothesis $p \sim q$ that $q \triangleq \perp$.

Induction step.

- $p \triangleq ((\alpha \cdot \pi); p') \oplus p''$. Then, $(p, \sigma_\alpha :: H, H') \xrightarrow{(\sigma_\alpha, \sigma_\pi)} (p', H, \sigma_\pi :: H')$ implies by the hypothesis $p \sim q$ that $(q, \sigma_\alpha :: H, H') \xrightarrow{(\sigma_\alpha, \sigma_\pi)} (q', H, \sigma_\pi :: H')$ and $p' \sim q'$. Recall that q is in n.f.; hence, by the shape of the semantic rules in Figure 4 it holds that $q \triangleq ((\alpha \cdot \pi); q') \oplus q''$. By the induction hypothesis, it holds that $p' \equiv q'$ hence, $(\alpha \cdot \pi); p'$ is a summand of q as well.
- Cases $p \triangleq (c; p') \oplus p''$ with $c ::= x?n \mid x!n \mid \mathbf{rcfg}_{x,n}$ follow in a similar fashion.

Hence, $p \equiv q \oplus p$ holds. The symmetric case $q \equiv p \oplus q$ follows the same reasoning.

We refer to [6] for the complete proofs and additional details.

4 A Framework for Safety

In this section we provide a language for specifying safety properties of DyNetKAT networks, together with a procedure for reasoning about safety in an equational fashion. Intuitively, safety properties enable specifying the absence of undesired network behaviours.

Definition 10 (Safety Properties - Syntax) *Let \mathcal{A} be an alphabet over letters of shape $\alpha \cdot \pi$ and $\mathbf{rcfg}_{x,p}$, with α and π ranging over complete tests and assignments, and $\mathbf{rcfg}_{x,p}$ ranging over reconfiguration actions. Safety properties are defined in the following fashion:*

$$\begin{aligned} act &::= \alpha \cdot \pi \mid \mathbf{rcfg}_{x,p} \quad (\alpha \cdot \pi, \mathbf{rcfg}_{x,p} \in \mathcal{A}) \\ regexp &::= true \mid act \mid \neg act \mid regexp + regexp \mid regexp \cdot regexp \mid \\ &\quad (regexp)^n \quad (\text{with } n \geq 1) \\ prop &::= [regexp]false \end{aligned}$$

A safety property specification $prop$ is satisfied whenever the behaviour encoded by $regexp$ should not be observed within the network. Regular expressions

$regexp$ are defined with respect to actions act : a flow of shape $\alpha \cdot \pi$ is the observable behaviour of a ($\text{NetKAT}^{\text{dup}}$) policy transforming a packet encoded by α into α_π , whereas $\mathbf{rcfg}_{x,p}$ corresponds to a reconfiguration step in a network. Recursively, a sum of regular expressions $regexp_1 + regexp_2$ encodes the union of the two behaviours, a concatenation of regular expressions $regexp_1 \cdot regexp_2$ encodes the behaviour of $regexp_1$ followed by the behaviour of $regexp_2$. A property of shape $[\neg a]false$, with $a \in \mathcal{A}$, states that the system cannot do anything apart from a as a first step. The property $[true]false$ states that no action can be observed in the network, whereas $[r^n]false$ encodes the repeated application of r for n times.

Note that $true$, negated expressions $\neg a$ and repetitions r^n are mere syntactic sugars of equivalent expressions free of these operations. Not surprisingly, “de-sugaring” ($ds(-)$) is defined as:

$$\begin{aligned}
 ds(true) &\triangleq \Sigma_{a \in \mathcal{A}} a \\
 ds(\neg a) &\triangleq \Sigma_{\substack{a_i \in \mathcal{A} \\ a_i \neq a}} a_i & ds(r^n) &\triangleq ds(\underbrace{r \cdot r \cdot \dots \cdot r}_{n \text{ times}}) \\
 ds(r_1 \cdot r_2) &\triangleq ds(r_1) \cdot ds(r_2) \text{ if } r_1 \cdot r_2 \text{ not de-sugared} \\
 ds(r_1 + r_2) &\triangleq ds(r_1) + ds(r_2) \text{ if } r_1 + r_2 \text{ not de-sugared} \\
 ds(r) &\triangleq r \text{ [otherwise]}
 \end{aligned}$$

The complete formal definition of the de-sugaring function is provided in [6].

Definition 11 (Safety Properties - Semantics) *Let \mathcal{A} be an alphabet over letters of shape $\alpha \cdot \pi$ and $\mathbf{rcfg}(\mathbf{x}, \mathbf{p})$, with α and π ranging over complete tests and assignments, and $\mathbf{rcfg}(\mathbf{x}, \mathbf{p})$ ranging over reconfiguration actions. We write w, w' for (non-empty) words with letters in \mathcal{A} (i.e., $w, w' \in \mathcal{A}^*$) and $|w|$ for the length of w . We write $w' \preceq w$ whenever w' is a prefix of w (including w).*

Let r be a de-sugared regular expression ($regexp$) as in Definition 10. We call head normal form (h.n.f.) of r , denoted by $hnf(r)$, the sum of words as above obtained by left-/right- distributing \cdot over $+$ in r , in the standard fashion. Note that such a h.n.f. always exists for r . Let Prop stand for the set of all properties as in Definition 10, in h.n.f.

The semantic map $\llbracket - \rrbracket : \text{Prop} \rightarrow \text{DyNetKAT}$ associates to each safety property in Prop a DyNetKAT expression as follows. Let Θ be the DyNetKAT policy (in normal form) encoding all possible behaviours over \mathcal{A} :

$$\Theta \triangleq \Sigma_{a \in \mathcal{A}}^\oplus (a; \perp \oplus a; \Theta)$$

Then:

$$\begin{aligned}
 \llbracket [\Sigma_{i \in I} w_i]false \rrbracket &\triangleq \Sigma_{\substack{w \in \mathcal{A}^* \\ |w| < M}}^\oplus \bar{w}; \perp \oplus \Sigma_{\substack{w \in \mathcal{A}^* \\ |w| = M}}^\oplus (\bar{w}; \perp \oplus \bar{w}; \Theta) \quad (4) \\
 w_i \in \mathcal{A}^* & \quad \forall i \in I : w_i \not\preceq w & \quad \forall i \in I : w_i \not\preceq w
 \end{aligned}$$

such that M is the length of the longest word w_i , with $i \in I$, and \bar{w} is a DyNetKAT-compatible term obtained from w where all letters have been sep-

arated by $;$ and inductively defined in the obvious way:

$$\bar{a} \triangleq a \quad (a \in \mathcal{A}) \qquad \overline{a \cdot w} \triangleq a; \bar{w} \quad (a \in \mathcal{A}, w \in \mathcal{A}^*)$$

The semantic map $\llbracket - \rrbracket$ is defined in accordance with the intuition provided in the beginning of this section. For instance, as shown in (4), if none of the sequences of steps w_i can be observed in the system, then the associated DyNetKAT term prevents the immediate execution of all w_i .

Typically, safety analysis is reduced to reachability analysis. In our context, a safety property is violated whenever the network system under analysis displays a (finite) execution that is not in the behaviour of the property. Thus, the aforementioned semantic map is based on traces (or words in \mathcal{A}^*) and is not sensitive to branching. This paves the way to reasoning about the satisfiability of safety properties in an equational fashion.

Definition 12 (Safe Network Systems) *Let E_{DNK}^{tr} stand for the equational axioms in Figure 6, including the additional axiom that enables switching from the context of bisimilarity to trace equivalence of DyNetKAT policies, namely: $p; (q \oplus r) \equiv p; q \oplus p; r$. Assume a specification given as the safety formula s and a network system implemented as the DyNetKAT policy i . We say that the network is safe whenever the following holds: $E_{DNK}^{tr} \vdash \llbracket s \rrbracket \oplus i \equiv \llbracket s \rrbracket$. In words: checking whether i satisfies s reduces to checking whether the trace behaviour of i is included into that of s .*

For an example, consider the firewall in Figure 1 and the corresponding encoding in Figure 3. Recall that reaching *int* from *ext* without observing a secure connection request is a faulty behaviour. This entails the safety formula s_n defined as $\llbracket (\neg \mathbf{rcfg}_{secConReq,1})^n \cdot (\alpha \cdot \pi) \rrbracket false$, for $n \in \mathbb{N}$, $\alpha \triangleq (port = ext)$ and $\pi \triangleq (port \leftarrow int)$. Therefore, checking whether the network is safe reduces to checking, for all $n \in \mathbb{N}$: $E_{DNK}^{tr} \vdash \llbracket s_n \rrbracket \oplus Init \equiv \llbracket s_n \rrbracket$. Note that, for a fixed n , the verification procedure resembles bounded model checking [4].

5 Implementation

In this section we describe our implementation for performing formal reasoning about dynamic updates in networks. We developed a prototype tool, called DyNetiKAT⁵, based on Maude [7], the NetKAT decision procedure [9] and Python [18] as a glue language. Our modular extension of the NetKAT semantics has allowed us to reuse the existing NetKAT tools while building our framework. In our prototype we focus on checking reachability and waypointing properties in a dynamic setting. We build upon the methods for checking reachability and waypointing properties in NetKAT [2]. For a reminder, in NetKAT, reachability and waypointing properties are characterised as follows: for reachability properties, an egress point *out* is reachable from an ingress point *in*, in the context of

⁵ <https://github.com/hcantunc/DyNetiKAT>

a switch policy p and topology t , whenever the following NetKAT equivalence holds: $in \cdot (p \cdot t)^* \cdot out \not\equiv \mathbf{0}$ (and vice-versa). For waypointing properties, an intermediate point w between in and out is considered a waypoint from in to out if all the packets from in to out go through w . Such a property is satisfied if the following NetKAT equivalence holds:

$$\begin{aligned} in \cdot (p \cdot t)^* \cdot out + in \cdot (-out \cdot p \cdot t)^* \cdot w \cdot (-in \cdot p \cdot t)^* \cdot out \\ \equiv in \cdot (-out \cdot p \cdot t)^* \cdot w \cdot (-in \cdot p \cdot t)^* \cdot out \end{aligned}$$

In order to utilise the NetKAT decision procedure for property checking we represent the properties given as regular expressions (as described in Section 4) as in the style of NetKAT properties. To this end, we introduced the operators $head(D)$, and $tail(D, R)$, where D is a DyNetKAT term and R is a set of terms of shape $\mathbf{rcfg}_{X,N}$. Intuitively, the operator $head(D)$ returns a NetKAT policy which represents the current configuration in the input D and the operator $tail(D, R)$ returns a DyNetKAT policy which is the sum of DyNetKAT policies inside D that appear after the synchronisation events in R . We utilise these operators in our procedure in the following fashion: for a given DyNetKAT term we apply our equational reasoning framework to unfold the expression and rewrite it into the normal form. Then, we extract the desired configurations by using the head and tail operators. After this step, the resulting expression is a purely NetKAT term and we utilise the NetKAT decision procedure for checking the desired properties.

For an example, consider the safety property $[(true)^n \cdot (\alpha \cdot \pi)]false$ as in Definition 10, and a network SDN . Note that for a given complete assignments there exists a corresponding complete test with the same values, e.g., the corresponding complete test for the complete assignment $f_0 \leftarrow v_0 \dots f_n \leftarrow v_n$ is $f_0 = v_0 \dots f_n = v_n$. In the remainder of this paper we write α_π to represent the corresponding complete tests of π . The property $[(true)^n \cdot (\alpha \cdot \pi)]false$ can be encoded in the style of NetKAT as follows:

$$\alpha \cdot head(\pi_n(SDN)) \cdot \alpha_\pi \equiv \mathbf{0} \tag{5}$$

$$\alpha \cdot head(tail(\pi_n(SDN), R)) \cdot \alpha_\pi \equiv \mathbf{0} \tag{6}$$

where R is the set of all synchronisation events in the network and $\pi_n(-)$ is the projection operator equationally defined in Figure 6. Note that in practice the parameter n in π_n is a fixed value specified by the user. Intuitively, (5) expresses that the initial configuration of the network is not able to transform the packets satisfying the predicate α such that they satisfy the predicate α_π and (6) expresses that this transformation is still not possible in the configurations after any sequence of synchronisation events. Formally, the operators $head$ and $tail$ are defined as follows:

$$\begin{array}{ll}
head(\perp) = \mathbf{0} & tail(\perp, R) = \perp \\
head(N; D) = N + head(D) & tail(N; D, R) = tail(D, R) \\
head(D \oplus Q) = head(D) + head(Q) & tail(D \oplus Q, R) = tail(D, R) \oplus tail(Q, R) \\
head(\mathbf{rcfg}_{X,N}; D) = \mathbf{0} & tail(\mathbf{rcfg}_{X,N}; D, R) = D \oplus tail(D, R) \text{ if } \mathbf{rcfg}_{X,Z} \in R \\
& tail(\mathbf{rcfg}_{X,N}; D, R) = \perp \text{ if } \mathbf{rcfg}_{X,N} \notin R
\end{array}$$

Note that we assume the DyNetKAT terms given as input to the operators *head* and *tail* do not contain terms of shape $x?q$ and $x!q$. This can be ensured by applying the restriction operator δ on the input terms.

For another example, consider again the stateful firewall in Figure 1 and the corresponding encoding in Figure 3. Recall that in this example reaching *int* from *ext* without observing a secure connection request constitutes a faulty behaviour. This property was formalized in Section 4 as follows: $[(-\mathbf{rcfg}_{secConReq,1})^n \cdot (\alpha \cdot \pi)]false$ where $\alpha \triangleq (port = ext)$ and $\pi \triangleq (port \leftarrow int)$. This property can be encoded in the NetKAT style as follows:

$$\alpha \cdot head(\pi_n(Init)) \cdot \alpha_\pi \equiv \mathbf{0} \quad (7)$$

$$\alpha \cdot head(tail(\pi_n(Init), R')) \cdot \alpha_\pi \equiv \mathbf{0} \quad (8)$$

where $R' = R \setminus \{\mathbf{rcfg}_{secConReq,1}\}$ and R is set of all synchronisation events in the network. Intuitively, the equivalence in (7) expresses that packets at port *ext* are not able to reach to port *int* in the initial configuration and (8) expresses that this behaviour is still not possible in the configurations after any sequence of synchronisation events apart from the $\mathbf{rcfg}_{secConReq,1}$ event.

Observe that the safety properties of Definition 10 are designed to capture unsafe flows. However, in a similar fashion, one can also define the syntax $\langle regexp \rangle true$ to express that a certain safe flow is possible and reason about it (possibly in combination with safety properties). For an example, consider the stateful firewall example and the property $\langle (\mathbf{rcfg}_{secConReq,1})^n \cdot (\alpha \cdot \pi) \rangle true$ where $\alpha \triangleq (port = ext)$ and $\pi \triangleq (port \leftarrow int)$. This property expresses that the flow from port *ext* to port *int* is possible after the event $\mathbf{rcfg}_{secConReq,1}$. This property can be encoded in the NetKAT style as follows:

$$\alpha \cdot head(tail(\pi_n(Init), R)) \cdot \alpha_\pi \neq \mathbf{0} \quad (9)$$

where $R = \{\mathbf{rcfg}_{secConReq,1}\}$.

We performed experiments on the FatTree [21] topologies to evaluate the performance of our implementation. FatTrees are hierarchical topologies which are very commonly used in data centers. Figure 7 illustrates a FatTree with 3 levels: core, aggregation and top-of-rack (ToR). The switches at each level contain a number of redundant links to the switches at the next upper level. The groups of ToR switches and their corresponding aggregation switches are called pods. For our experiments, we generated 6 FatTrees that grow in size and achieve a maximum size of 1344 switches. For these networks we computed a shortest path forwarding policy between all pairs of ToR switches. The number

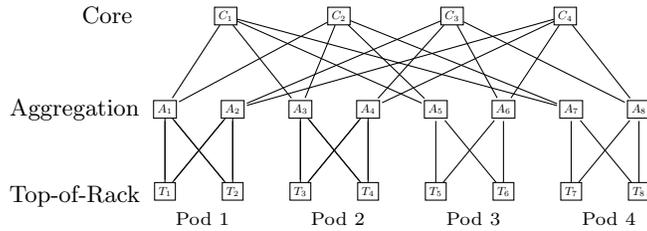


Fig. 7: A FatTree Topology

of switches in the ToR layer is set to $k^3/4$ where k is the number of pods in the network.

We check certain dynamic properties on these networks and assess the time performance of our tool. We consider a scenario which involves two ToR switches T_a and T_b that reside in different pods. Initially, all the packets from T_a to T_b traverse through a firewall A_x in the aggregation layer which filters SSH packets. The controller then decides to shift the firewall from A_x to another switch $A_{x'}$ in the aggregation layer. For this purpose, the controller updates the corresponding aggregation and core layer switches which results in a total of 4 updates. The properties that we check in this scenario are as follows: (i) At any point while the controller is performing the updates, non-SSH packets from T_a can always reach to T_b . (ii) At any point while the controller is performing the updates, SSH packets from T_a can never reach to T_b . (iii) After all the updates are performed, $A_{x'}$ is a waypoint between T_a and T_b .

We conducted the experiments on a computer running Ubuntu 20.04 LTS with 16 core 2.4GHz Intel i9-9980HK processor and 64 GB RAM. The results of these experiments are displayed in Figure 8. We report the preprocessing time, the time taken for checking properties (i), (ii), and (iii) individually (referred to as Reachability-I, Reachability-II, and Waypointing, respectively), and also time taken to check all the properties in parallel (referred to as All Properties). The reported times are the average of 10 runs.

The results indicate that preprocessing step is a non-negligible factor that contributes to overall time. However, preprocessing is independent of the property that is being checked and this procedure only needs to be done once for a given network. After the preprocessing step, the individual properties can be checked in less than 2 seconds for networks with less than 100 switches. For larger networks with sizes up to 931 and 1344 switches, the individual properties can be checked in a maximum of 5 minutes and 11 minutes, respectively. It can be observed that checking for the property (iii) can take more than twice as much time as checking for the properties (i) and (ii). In the experiments where we check all the properties in parallel, we allocated one thread for each property. In this setting, checking all the properties in parallel introduced 24% overhead on average. An important observation in the results of these experiments is that after the preprocessing is performed, on average 87% of the running times are

spent in the NetKAT decision procedure and this step becomes the bottleneck in analysing larger networks.

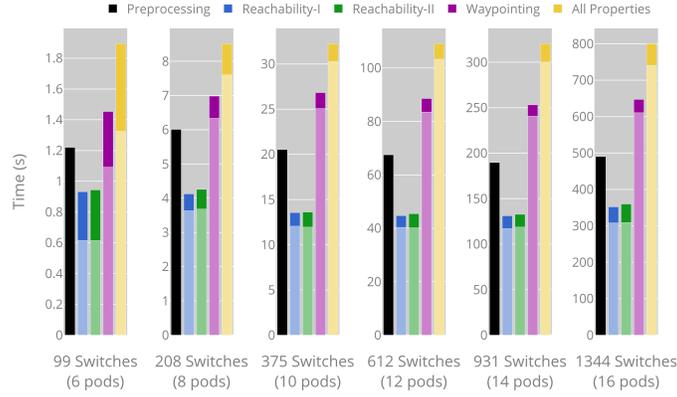


Fig. 8: Results of FatTree experiments. Light coloured areas indicate the time spent in the NetKAT decision procedure and solid coloured areas indicate the time spent in our equational reasoning framework.

6 Conclusions

We developed a language, called DyNetKAT, for modelling and reasoning about dynamic reconfigurations in Software Defined Networks. Our language builds upon the concepts, syntax, and semantics of NetKAT and hence, provides a modular extension and makes it possible to reuse the theory and tools of NetKAT. We define a formal semantics for our language and provide a sound and ground-complete axiomatisation. We exploit our axiomatisation to analyse reachability properties of dynamic networks and show that our approach is indeed scalable to networks with hundreds of switches.

Our language builds upon the assumption that control plane updates interleave with data plane packet processing in such a way that each data plane packet sees one set of flow tables throughout their flight in the network. This assumption is inspired by the framework put forward by Reitblatt et al. [16] and is motivated by the requirement to design a modular extension on top of NetKAT. However, we have experimented with a much smaller-stepped semantics in which the control plane updates can have a finer interleaving with in-flight packet moves. This alternative language breaks the hierarchy with NetKAT and a naive treatment of this alternative semantics will lead to much larger state-spaces. We would like to investigate this small-step semantics and efficient analysis techniques for it further.

References

1. Luca Aceto, Bard Bloom, and Frits W. Vaandrager. Turning SOS rules into equations. *Inf. Comput.*, 111(1):1–52, 1994. doi:10.1006/inco.1994.1040.
2. Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: semantic foundations for networks. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 113–126. ACM, 2014. doi:10.1145/2535838.2535862.
3. Jos C. M. Baeten and W. P. Weijland. *Process algebra*, volume 18 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1990.
4. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
5. Ryan Beckett, Michael Greenberg, and David Walker. Temporal netkat. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 386–401. ACM, 2016. doi:10.1145/2908080.2908108.
6. G. Caltais, H. Hojjat, M. R. Mousavi, and H. C. Tunc. DyNetKAT: An algebra of dynamic networks. *CoRR*, abs/2102.10035, 2021.
7. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Full Maude: Extending Core Maude. In Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors, *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*, pages 559–597. Springer, 2007. doi:10.1007/978-3-540-71999-1_18.
8. Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic NetKAT. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 282–309. Springer, 2016. doi:10.1007/978-3-662-49498-1_12.
9. Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A Coalgebraic Decision Procedure for NetKAT. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 343–355. ACM, 2015. doi:10.1145/2676726.2677011.
10. Tobias Kappé, Paul Brunet, Alexandra Silva, Jana Wagemaker, and Fabio Zanasi. Concurrent Kleene Algebra with Observations: from Hypotheses to Completeness. *CoRR*, abs/2002.09682, 2020. URL: <https://arxiv.org/abs/2002.09682>, arXiv:2002.09682.
11. Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russell J. Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 59–72. USENIX Association, 2015. URL: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kim>.

12. Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerný. Event-driven network programming. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 369–385. ACM, 2016. doi:10.1145/2908080.2908097.
13. Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebraic Methods Program.* 60-61: 195-228, 2004. doi.org/10.1016/j.jlap.2004.03.008
14. Mohammad Reza Mousavi, Michel A. Reniers, and Jan Friso Groote. Notions of bisimulation and congruence formats for SOS with data. *Information and Computation*, 200(1):107 – 147, 2005. doi.org/10.1016/j.ic.2005.03.002.
15. Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 519–531. USENIX Association, 2014. URL: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/nelson>.
16. Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In Lars Eggert, Jörg Ott, Venkata N. Padmanabhan, and George Varghese, editors, *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*, pages 323–334. ACM, 2012. doi:10.1145/2342356.2342427.
17. Alexandra Silva. Models of Concurrent Kleene Algebra. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, page 516. EasyChair, 2020. URL: <https://easychair.org/publications/paper/6C8R>.
18. Guido van Rossum. Python programming language. In Jeff Chase and Srinivasan Seshan, editors, *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007*. USENIX, 2007.
19. Alexander Vandenbroucke and Tom Schrijvers. Pλwnk: functional probabilistic netkat. *Proc. ACM Program. Lang.*, 4(POPL):39:1–39:27, 2020. doi:10.1145/3371107.
20. Jana Wagemaker, Paul Brunet, Simon Docherty, Tobias Kappé, Jurriaan Rot, and Alexandra Silva. Partially Observable Concurrent Kleene Algebra. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPICs*, pages 20:1–20:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.CONCUR.2020.20.
21. Al-Fares, Mohammad and Loukissas, Alexander and Vahdat, Amin. A Scalable, Commodity Data Center Network Architecture. *ACM SIGCOMM Comput. Commun. Rev.* 38, 4, 63-74, 2008. doi:10.1145/1402946.1402967.