Concurrent NetKAT

Modeling and analyzing stateful, concurrent networks

Wagemaker, J.; Foster, N.; Kappé, T.; Kozen, D.; Rot, J.; Silva, A.

DOI
10.48550/arXiv.2201.10485
10.1007/978-3-030-99336-82_1

Publication date
2022

Document Version
Final published version

Published in
Programming Languages and Systems

License
CC BY

Concurrent NetKAT

Modeling and analyzing stateful, concurrent networks

Jana Wagemaker[1] ✉, Nate Foster[2], Tobias Kappé[3], Dexter Kozen[2], Jurriaan Rot[1], and Alexandra Silva[2]

[1] Radboud University, Nijmegen, The Netherlands
`Jana.Wagemaker@ru.nl`
[2] Cornell University, Ithaca, New York, USA
[3] ILLC, University of Amsterdam, The Netherlands

Abstract. We introduce Concurrent NetKAT (CNetKAT), an extension of NetKAT with operators for specifying and reasoning about concurrency in scenarios where multiple packets interact through state. We provide a model of the language based on partially-ordered multisets (pomsets), which are a well-established mathematical structure for defining the denotational semantics of concurrent languages. We provide a sound and complete axiomatization of this model, and we illustrate the use of CNetKAT through examples. More generally, CNetKAT can be understood as an algebraic framework for reasoning about programs with both local state (in packets) and global state (in a global store).

Keywords: Concurrent Kleene algebra, NetKAT, completeness, concurrency

1 Introduction

Kleene algebra (KA) is a well-studied formalism [20,23,34,8] for analyzing and verifying imperative programs. Over the past few decades, various extensions of KA have been proposed for modeling increasingly sophisticated scenarios. For example, Kleene algebra with tests (KAT) [21] models conditional control flow while NetKAT [3,10] models behaviors in packet-switched networks.

A key limitation of NetKAT, however, is that the language is stateless and sequential. It cannot model programs composed in parallel, and it offers no way to reason algebraically about the effects induced by multiple concurrent packets. Meanwhile, the software-defined networking (SDN) paradigm has evolved to include richer functionality based on stateful processing including data aggregation and dynamic routing. In languages like P4 [4], issues of concurrency arise because the semantics depends on the order that packets are processed.

Given this context, it is natural to wonder we can add concurrency to NetKAT while retaining the elegance of the underlying framework. In this paper, we answer this question in the affirmative, by developing CNetKAT. However, to do this, we must overcome several challenges. A first hurdle is that networks exhibit many different forms of concurrent behavior. The most obvious source

I. Sergey (Ed.): ESOP 2022, LNCS 13240, pp. 575–602, 2022.
https://doi.org/10.1007/978-3-030-99336-8_21

of concurrency arises when multiple packets are processed by different devices. In these situations, certain packets may cause changes in forwarding behavior by modifying global state variables on switches. However, there is also concurrency within individual devices: a high-speed switching chip often has multiple pipelines, each with multiple stages of match-action tables and stateful registers. The tables can be programmed to act concurrently on (parts of) a single packet, and the pipelines also act concurrently on multiple packets.

Another hurdle is that it is not entirely clear how to simultaneously extend KA with networking features and concurrency. Orthogonal to the development of NetKAT, the issue of adding concurrency to KA has been researched extensively, starting with concurrent Kleene algebra (CKA) [13,25,26,17]. However, the combination of concurrency from CKA and tests from KAT is not straightforward—see, e.g. [14,15,16]—which motivated the development of partially-observable concurrent Kleene algebra (POCKA) [37]. In POCKA, a single thread only has *partial* view of the state. Hence, when evaluating control guards, a thread makes *observations* about the machine state, rather than definitive tests. This allows for fine-grained reasoning about concurrent programs with variables, conditionals, loops, and imperative statements that manipulate a shared global memory.

In this work, we use POCKA as a basis for designing a language with state and concurrent threads, which we combine with a multi-packet extension of NetKAT. The resulting language, Concurrent NetKAT (CNetKAT), models the behavior of packets in a network that communicate through a shared global state, and addresses the fundamental and non-trivial question of how to combine concurrency and the interaction between local and global state within KA.

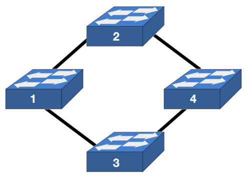
Overall, the contributions of the paper are as follows:

1. We present the design of the CNetKAT language (§3). The semantics combines the language models of NetKAT and POCKA, incorporating pomsets that record the evolution of the global state (as in POCKA) as well as sets of (output) packets (as in NetKAT).
2. We develop a sound and complete axiomatization of CNetKAT (§4).
3. We illustrate the applicability of CNetKAT for modeling and analyzing concurrent network behaviors through case studies and examples (§2 and §5).

The next section contains an overview of the challenges in the design of extending NetKAT with multiple packets, global state, and concurrency, as well as a glimpse of how to use the language in a practical example.

2 Overview

CNetKAT models the behavior of two basic entities: the packets being routed through the network, and a global store, which may be accessed by the network as it processes the packets. These elements give rise to two kinds of basic programs. On the one hand, *basic packet programs*—imported from NetKAT [3]—include tests ($f_i{=}n$) and modifications ($f_i{\leftarrow}n$) of packet fields $f_1, \dots, f_N$. Examples of fields are sw, denoting the switch of the packet in the network, and tag, denoting



$$\begin{aligned} p_1 &\triangleq \mathsf{sw} = 1\,;((v = 1\,;\mathsf{tag} = \spadesuit\,;\mathsf{sw}{\leftarrow}2) \\ &\qquad\qquad \parallel (\mathsf{tag} = \heartsuit\,;\mathsf{sw}{\leftarrow}3\,;v{\leftarrow}1)) \\ p_2 &\triangleq \mathsf{sw} = 2; \mathsf{sw}{\leftarrow}4 \\ p_3 &\triangleq \mathsf{sw} = 3; \mathsf{sw}{\leftarrow}4 \\ p_4 &\triangleq \mathsf{sw} = 4 \\ \\ p &\triangleq \; v{\leftarrow}0\,;(p_1 \parallel p_2 \parallel p_3 \parallel p_4)^* \end{aligned}$$

Fig. 1: Running example

the type of a packet. In general, we expect packets to have fields for a collection of standard attributes; unused fields may be populated with a dummy value.

On the other hand, *basic state programs* include observations[4] $(v_i{=}n)$, modifications $(v_i{\leftarrow}n)$ and a copy operation $(v_i{\leftarrow}v_j)$ on state variables $v_1, \ldots, v_M$. It will always be clear from context whether an action concerns a state or field variable. CNetKAT also includes a primitive program a for any set of packets a, which is useful for specifying the set of packets currently being processed.

Remark 1. We could augment the set of primitives with features such as general expressions in assignments. However, to keep things simple, we will only consider these primitives, which are already rich enough to describe non-trivial behaviors.

CNetKAT programs are composed using sequential composition (';'), iteration ('$*$'), and non-deterministic choice ('$+$'), similar to NetKAT. In addition, CNetKAT programs may use the parallel composition operator ('$\parallel$').

The full syntax of CNetKAT is given in Figure 2. Before giving a precise account of the semantics, we will go over some simple example programs.

Example 1 (Packet forwarding). Consider the network depicted on the left in Figure 1. Similar to NetKAT, we assume packet movement and variable assignments are instantaneous. Suppose there are two packet types: $\spadesuit$ and $\heartsuit$. We want to write a program that transfers packets from node 1 to node 4 by sending $\spadesuit$ via node 2, and $\heartsuit$ via node 3. The program running in switch 1 could be

$$p_1 := \mathsf{sw}{=}1\,;((\mathsf{tag}{=}\spadesuit\,;\mathsf{sw}{\leftarrow}2) \parallel (\mathsf{tag}{=}\heartsuit\,;\mathsf{sw}{\leftarrow}3))$$

This program first filters out the packets at switch 1. Next, it launches two parallel threads, both of which receive a copy of the incoming packets. The first thread filters out packets of type $\spadesuit$ and forwards them to switch 2, while the second thread filters out packets of type $\heartsuit$, forwarding them to switch 3.

We can write programs p_2, p_3 and p_4 for the other switches as well, and then compose all of those in parallel to obtain a program for the entire network.

Remark 2. Instant packet movement is not baked into CNetKAT, but rather a consequence of modeling packet location using the field sw. A more advanced

[4] Intuitively, these are tests on the state that can be understood as observing the part of the global state containing the variable, hence the terminology.

model could use an additional field to mark a packet as being "in-flight" until it reaches the next hop. Here, we opt for the simpler model.

Example 2 (Global behavior). CNetKAT programs can read and write to a global store, letting earlier actions on packets affect later decisions. For instance, suppose we need ♠ packets to be forwarded only if a ♡ packet already visited switch 3. We can use a global variable v to implement this stateful behavior, writing:

$$\mathsf{sw}{=}1 \,;\, ((v{=}1 \,;\, \mathsf{tag}{=}\spadesuit \,;\, \mathsf{sw}{\leftarrow}2) \parallel (\mathsf{tag}{=}\heartsuit \,;\, \mathsf{sw}{\leftarrow}3 \,;\, v{\leftarrow}1))$$

We can program the other switches with p_i, as shown in Figure 1.

Remark 3 (Concurrency and state). Actions involving global variables are more subtle than those that concern packet fields, due to concurrent threads accessing the global store. For instance, we can write the program $v{\leftarrow}1 \,;\, v{=}2$, which first sets v to 1 and then asserts that v should have value 2. This may seem inconsistent; however, there may be valid ways of executing this program if there are other threads that change the value of v from 1 to 2 between the assignment $v{\leftarrow}1$ and the assertion $v{=}2$. This possibility makes defining a compositional semantics somewhat tricky, as we will discuss below.

Semantics of CNetKAT programs. A packet π is a record of fields $f_1, \ldots, f_N$. We write $\pi(\mathsf{sw})$ for the value of sw in π and $\pi[1/\mathsf{sw}]$ for the packet obtained after updating the value of sw to 1. We denote the set of packets by Pk.

The semantics of a CNetKAT program is represented as a function that takes a set of packets, potentially located in different nodes in the network, and returns a set of possible behaviors that those input packets might produce. More precisely, the semantics function has type $[\![-]\!]\colon 2^{\mathsf{Pk}} \to 2^{\mathcal{P}om \cdot 2^{\mathsf{Pk}}}$. Here, $\mathcal{P}om$ is the set of *pomsets* [12,11], which can be thought of as structures that record the causal order between concurrent events (details appear in Section 3.1). An element $\mathbf{u} \cdot b \in [\![p]\!](a)$ means "there is an execution of p that changes the global variables according to $\mathbf{u}$, and the set of output packets produced is b".[5]

The semantics is defined in Figure 3. For instance, a packet filter $(f{=}n)$ takes a set of packets a and returns $\{\mathbf{1} \cdot a(f{=}n)\}$, where $a(f{=}n)$ contains all packets in a where f has value n and $\mathbf{1}$ is the pomset representing that the global state did not change. A modification $(f{\leftarrow}n)$ takes a set of input packets a and returns $\{\mathbf{1} \cdot a(f \leftarrow n)\}$, where $a(f \leftarrow n) = \{\pi[n/f] : \pi \in a\}$. These two basic packet actions manipulate the *local state* of the program.

On the global state we have observations of the form $(v{=}n)$ and modifications $(v{\leftarrow}n)$, $(v{\leftarrow}v')$. Each gives rise to a pair in the semantics—$\{v = n \cdot a\}$, $\{(v{\leftarrow}n) \cdot a\}$, $\{(v{\leftarrow}v') \cdot a\}$—in which the input set of packets a is returned as output and the assertion or modification is recorded in the pomset.

Lastly, the primitive $a \in 2^{\mathsf{Pk}}$ is useful for writing specifications. This program copies the set of packets a into the global pomset. We will see that this is useful for checking inclusion of certain behaviors in a program's semantics, and in the

[5] We use the notation $\cdot$ to denote pairs: $\mathbf{u} \cdot b$ denotes the pair $(\mathbf{u}, b)$.

Syntax

Values	$\mathsf{Val} \ni n$	$::= 0 \mid 1 \mid 2 \mid \cdots$	
Packet Fields	$\mathsf{Fld} \ni f$	$::= f_1 \mid \cdots \mid f_k$	
Packets	$\mathsf{Pk} \ni \pi$	$::= \{f_1 = n_1, \ldots\}$	
Packet Sets	$2^{\mathsf{Pk}} \ni a, b$		
Packet predicates	$\mathcal{B} \ni t, u$	$::=$	
	$\mid$	drop	*False*
	$\mid$	pass	*True*
	$\mid$	$f{=}n$	*Field Test*
	$\mid$	$t \vee_{\mathcal{B}} u$	*Disjunction*
	$\mid$	$t \wedge_{\mathcal{B}} u$	*Conjunction*
	$\mid$	$\neg t$	*Negation*
State obs.	$\mathcal{O} \ni o, o'$	$::=$	
	$\mid$	$\bot$	*Inconsistent*
	$\mid$	$\top$	*Neutral*
	$\mid$	$v{=}n$	*State test*
	$\mid$	$o \vee o'$	*Union*
	$\mid$	$o \wedge o'$	*Intersection*
	$\mid$	$\overline{o}$	*Complement*

State Fields	$\mathsf{Var} \ni v$	$::= v_1 \mid \cdots \mid v_i$	
Global State	$\mathsf{St} \ni \alpha, \beta$	$::= \mathsf{Var} \rightharpoonup \mathsf{Val}$	
State Act actions	$\ni e$	$::=$	
	$\mid$	$v{\leftarrow}n$	*Change*
	$\mid$	$v{\leftarrow}v'$	*Copy*
Programs	$\mathsf{Prg} \ni p, q$	$::=$	
	$\mid$	abort	*Abort*
	$\mid$	skip	*Skip*
	$\mid$	t	*Packet Filter*
	$\mid$	o	*State Obs*
	$\mid$	$f{\leftarrow}n$	*Packet Action*
	$\mid$	e	*State Action*
	$\mid$	dup	*Duplicate*
	$\mid$	$p + q$	*Choice*
	$\mid$	$p \mathbin{;} q$	*Sequence*
	$\mid$	$p \parallel q$	*Parallel*
	$\mid$	p^*	*Iteration*
	$\mid$	a	*Packet Sets*

Fig. 2: CNetKAT syntax. We highlight constructs not in NetKAT.

proof of completeness. Formally, the behavior of a on any input set b is $\{a \cdot b\}$, where a is the global state pomset with one node labeled by a.

To construct more complicated programs, we can combine the basic elements above using operators from Kleene algebra. For instance, $p+q$ is a program that represents a non-deterministic choice between p and q. Its semantics is obtained by taking the union of sets produced by both p and q on the input packets. We can also compose programs sequentially using $p\,;q$, where we first apply p to the input packets and then q to all sets of packets produced by p, and we compose the corresponding global pomsets sequentially. We can iterate a program finitely many times using p^*. Lastly, we can combine programs with a parallel operator, $p \parallel q$, which denotes a program that, on input a, executes both p and q on a, and then combines the results: the pomsets denoting the global components are composed in parallel, and the corresponding sets of output packets joined.

Remark 4 (Concurrency and state, continued). Note that statements observing or modifying global variables are stored in the pomsets but not executed, that is, we do not actually *check* immediately whether v is indeed 1 but rather simply record it. This may seem like an odd choice at first: why does the semantics not also keep a record of the global store? The reason is related to Remark 3.

Consider the program $q = (v{=}0) \,;(v{=}1)$, which asserts that v has value 0, and then that it has value 1. In isolation, q does not have any valid behavior, as it sequentially executes two tests that cannot be valid without intermediate intervention. However, the program $q \parallel (v{\leftarrow}1)$ *does* have valid behavior on some

$$\boxed{[\![p]\!] : 2^{\mathsf{Pk}} \to 2^{\mathfrak{Pom}(\mathsf{St}\cup\mathsf{Act}\cup 2^{\mathsf{Pk}})\cdot 2^{\mathsf{Pk}}}}$$

Semantics

$$\begin{aligned}
[\![p]\!](\varnothing) &\triangleq \{1\cdot\varnothing\}\\
[\![\mathsf{abort}]\!](a) &\triangleq \varnothing\\
[\![\mathsf{skip}]\!](a) &\triangleq \{\mathbf{1}\cdot a\}\\
[\![t]\!](a) &\triangleq \{\mathbf{1}\cdot [\![t]\!]_{\mathcal{B}}(a)\}\\
[\![f\!\leftarrow\! n]\!](a) &\triangleq \{\mathbf{1}\cdot a(f\leftarrow n)\}\\
[\![b]\!](a) &\triangleq \{b\cdot a\}\\
[\![\mathsf{dup}]\!](a) &\triangleq \{a\cdot a\}\\
[\![p+q]\!](a) &\triangleq [\![p]\!](a)\cup[\![q]\!](a)
\end{aligned}$$

$$\begin{aligned}
[\![o]\!](a) &\triangleq \mathsf{St}^*\odot[\![o]\!]_{\mathcal{O}}\odot\mathsf{St}^*\times\{a\}\\
[\![e]\!](a) &\triangleq \mathsf{St}^*\odot\{e\}\odot\mathsf{St}^*\times\{a\}\\
[\![p\,;q]\!](a) &\triangleq \left\{(\mathbf{u}\cdot\mathbf{v})\cdot b \;\middle|\; \begin{array}{l}\mathbf{u}\cdot a'\in[\![p]\!](a),\\ \mathbf{v}\cdot b\in[\![q]\!](a')\end{array}\right\}\\
[\![p\parallel q]\!](a) &\triangleq \left\{(\mathbf{u}\parallel\mathbf{v})\cdot(b\cup c) \;\middle|\; \begin{array}{l}\mathbf{u}\cdot b\in[\![p]\!](a),\\ \mathbf{v}\cdot c\in[\![q]\!](a)\end{array}\right\}\\
[\![p^*]\!](a) &\triangleq \bigcup\Big\{[\![\underbrace{p\cdots p}_{n\text{ times}}]\!](a) : n\in\mathbb{N}\Big\}
\end{aligned}$$

Predicates

$$\begin{aligned}
[\![t]\!]_{\mathcal{B}}(a) &: 2^{\mathsf{Pk}}\\
[\![\mathsf{drop}]\!]_{\mathcal{B}}(a) &\triangleq \varnothing\\
[\![\mathsf{pass}]\!]_{\mathcal{B}}(a) &\triangleq a\\
[\![f=n]\!]_{\mathcal{B}}(a) &\triangleq a(f=n)\\
[\![t\vee_{\mathcal{B}} u]\!]_{\mathcal{B}}(a) &\triangleq [\![t]\!]_{\mathcal{B}}(a)\cup[\![u]\!]_{\mathcal{B}}(a)\\
[\![t\wedge_{\mathcal{B}} u]\!]_{\mathcal{B}}(a) &\triangleq [\![t]\!]_{\mathcal{B}}(a)\cap[\![u]\!]_{\mathcal{B}}(a)\\
[\![\neg t]\!]_{\mathcal{B}}(a) &\triangleq a\setminus[\![t]\!]_{\mathcal{B}}(a)
\end{aligned}$$

Observations

$$\begin{aligned}
[\![o]\!]_{\mathcal{O}} &: 2^{\mathsf{St}}\\
[\![\bot]\!]_{\mathcal{O}} &\triangleq \varnothing\\
[\![\top]\!]_{\mathcal{O}} &\triangleq \mathsf{St}\\
[\![v=n]\!]_{\mathcal{O}} &\triangleq \{\alpha\in\mathsf{St}\mid\alpha(v)=n\}\\
[\![o\vee o']\!]_{\mathcal{O}} &\triangleq [\![o]\!]_{\mathcal{O}}\cup[\![o']\!]_{\mathcal{O}}\\
[\![o\wedge o']\!]_{\mathcal{O}} &\triangleq [\![o]\!]_{\mathcal{O}}\cap[\![o']\!]_{\mathcal{O}}\\
[\![\overline{o}]\!]_{\mathcal{O}} &\triangleq \bigcup\{Z\in\mathcal{P}_{\leq}(\mathsf{St})\mid[\![o]\!]_{\mathcal{O}}\cap Z=\varnothing\}
\end{aligned}$$

Filtering, updates and downwards closure $a\in 2^{\mathsf{Pk}}, Z\subseteq\mathsf{St}$

$$\begin{array}{ll}
a(f=n)=\{\pi\in a\mid\pi(f)=n\} & a(f\leftarrow n)=\{\pi[n/f]\mid\pi\in a\}\\
\alpha\leq\beta \iff \mathsf{domain}(\beta)\subseteq\mathsf{domain}(\alpha)\;\wedge & \forall x\in\mathsf{domain}(\beta).\ \alpha(x)=\beta(x)\\
Z_{\leq}=\{\alpha\mid\exists\beta\in Z \text{ s.t } \alpha\leq\beta\} & \mathcal{P}_{\leq}(\mathsf{St})=\{Z\mid Z\subseteq\mathsf{St}\wedge Z=Z_{\leq}\}
\end{array}$$

Fig. 3: CNetKAT semantics. Pairs $\mathbf{u}\cdot b$ in $[\![p]\!](a)$ indicate that the program p takes input a and the global state change induced by p is encoded in $\mathbf{u}$ and constrains the final packet set b. We overload $\cdot$ for sequential composition of pomsets and pairs, while $\odot$ is the usual lifting from pomsets to languages.

interleavings—namely the ones where the assignment $v\!\leftarrow\!1$ is scheduled between the two tests. It stands to reason that a compositional semantics of such programs should include traces with such local inconsistencies, as they may be explained by actions taken by other programs running in parallel [37]. For CNetKAT, this is accomplished by placing the observations and modifications in the pomset.

This leaves us with the question of how to obtain the semantics of a program in isolation. We take a page from POCKA [37], which uses the set of *guarded pomsets* to filter out the pomsets sensible in isolation; details appear in §5.

One final modification is needed to obtain the CNetKAT semantics from $[\![-]\!]$. The idea is to allow interleaving between parallel threads [13]. This is accomplished by adding to the semantics all pomsets in which events are "more ordered" than the ones already present in $[\![-]\!]$. We denote this closed semantics by $[\![-]\!]{\downarrow}$; a precise definition is given in §3.

Recording local behavior To apply CNetKAT to various verification tasks, we sometimes need to take snapshots of the local state at different points. For example, if we want to argue that ♡ packets arrived at switch 3 before ♠ packets arrived at switch 2, we need more than the information about inputs and outputs that have occurred so far. We therefore have to extend the language with an operator comparable to dup in NetKAT. On input a, the semantics of the dup operator is the set $\{a \cdot a\}$, where the first component is a single node pomset labeled with set of packets a.[6] By recording packets inside the pomset, information about changes to packets also contains their relation to changes to global variables during the execution. Hence, using dup, we can infer causality relations between local and global state changes.

The programs p_1, p_2, p_3 and p_4 used in our running example (see Figure 1) can be instrumented with a dup on every entry to and exit from a switch. This encodes extra information in the semantics that can be used for reasoning about packet-forwarding paths as well as global state changes.

$$\begin{aligned}
p_1 &\triangleq \mathsf{sw} = 1 \mathbin{;} \mathsf{dup} \mathbin{;} ((v = 1 \mathbin{;} \mathsf{tag} = ♠ \mathbin{;} \mathsf{dup} \mathbin{;} \mathsf{sw}{\leftarrow}2 \mathbin{;} \mathsf{dup}) \\
&\qquad\qquad\qquad\qquad \parallel (\mathsf{tag} = ♡ \mathbin{;} \mathsf{dup} \mathbin{;} \mathsf{sw} \leftarrow 3 \mathbin{;} \mathsf{dup} \mathbin{;} v \leftarrow 1)) \\
p_2 &\triangleq \mathsf{sw} = 2 \mathbin{;} \mathsf{dup} \mathbin{;} \mathsf{sw}{\leftarrow}4 \mathbin{;} \mathsf{dup} \\
p_3 &\triangleq \mathsf{sw} = 3 \mathbin{;} \mathsf{dup} \mathbin{;} \mathsf{sw}{\leftarrow}4 \mathbin{;} \mathsf{dup} \\
p_4 &\triangleq \mathsf{sw} = 4 \mathbin{;} \mathsf{dup}
\end{aligned}$$

The overall program of the running example then becomes

$$p \triangleq v{\leftarrow}0 \mathbin{;} (p_1 \parallel p_2 \parallel p_3 \parallel p_4)^*$$

where the global variable v is initialized to 0, and the programs p_1, p_2, p_3, p_4 are executed in parallel, performing the actions of each individual switch. The Kleene star ensures that the packets may take multiple hops through the network, eventually reaching their final destination (switch 4).

Remark 5. If a dup occurs in parallel to other threads, then these other parallel threads can only change the exact place of the dup-recording in the pomset via possible interleavings, but not influence its content.

Remark 6. We model the collection of in-flight packets as a set, as opposed to e.g. a partially ordered set encoding their order of arrival. This is an abstraction of our framework. Not putting an order on packets simplifies the algebraic presentation and has the advantage that it enables modeling of switches that reorder packets without an additional primitive. If the order of packets is important, information about this order can be extracted from the semantics. In particular, when packets were forwarded can be deduced by inspecting the sets of packets recorded in the pomset component using dup.

Two differences between CNetKAT and NetKAT Readers familiar with NetKAT might wonder why Example 1 uses $\parallel$ instead of $+$ to compose the

[6] We overload 'a' as a set of packets, a programming primitive and a label used in pomsets, but it always denotes a set of packets in the latter two uses as well.

branches of p_1. The reason is that in CNetKAT, $\|$ is interpreted as multicast and $+$ is interpreted as non-deterministic composition. In NetKAT, programs act on a single input packet, so these coincide. But in CNetKAT, programs act on multiple packets concurrently, so they must be distinguished.

To illustrate the difference, consider wanting to filter the input packets so that only those where field f has value n or field g has value m remain. In NetKAT, we can use the program $f{=}n + g{=}m$, which can be understood in two different ways. First, we can think of it as using (angelic) non-determinism to select a test, yielding $\{\pi\}$ if at least one test passes and $\varnothing$ if both tests fail. Alternatively, we can think of it as using multicast to copy the input to both $f{=}n$ and $g{=}m$, then using the tests to perform the required filtering, and finally taking the union of the resulting sets. In NetKAT, the net effect of both interpretations is identical, so multicast and non-determinism can be identified semantically.

However, when we generalize to *sets* of packets, it is natural to expect that processing a set a with $f{=}n$ followed by $g{=}m$ would yield the subset of a where each packet satisfies at least one of the tests. Operationally, processing a using these programs could be realized by making two copies of a, then using the tests to perform the required filtering, and taking the union of the resulting sets. This is reflected in the semantics: $[\![f{=}m \parallel g{=}n]\!](a) = \{\mathbf{1} \cdot (a(f = m) \cup a(g = n))\}$, where we get a single pair in the output. If instead we non-deterministically choose between the tests, the result would be the subset where $f = n$ *or* the subset where $g = m$. Indeed, we have that $[\![f{=}m + g{=}n]\!](a) = \{\mathbf{1} \cdot a(f = m), \mathbf{1} \cdot a(g = n)\}$. Hence, multicast and non-determinism can no longer be identified in the context of multiple packets. For readers familiar with NetKAT, this means that the Boolean disjunction $\vee$ is now identified with $\|$ rather than $+$.

Lastly, we highlight that CNetKAT's dup is fundamentally different from NetKAT's dup, which just records versions of the packet during execution. In CNetKAT, dup does two things: it implements the same functionality as in NetKAT, but also structures the recording of packets inside the pomset.

Proving properties with CNetKAT In §5, we analyze the behavior of the running example in detail and show how to filter out the behaviors of p that can be obtained when it is run *in isolation*. In this overview, we establish a simpler property: namely, that p exhibits executions where the packets were at switch 3 before they were at switch 2. We first argue this using the denotational semantics and then illustrate how we can establish the same fact with axiomatic reasoning.

Recall a pomset accounts for events and the ordering between them. In the following examples, we will depict pomsets as a graph with nodes labeled by state actions, observations and sets of packets, and the ordering indicated by arrows. For instance, $a \to b$ means that a happened before b.

We evaluate p on input $\{\heartsuit, \spadesuit\}$, where both packets start at switch 1. In the closed semantics $[\![p]\!]{\downarrow}\,(\{\heartsuit, \spadesuit\})$ we find the following pomset (the $\cdots$ indicate that the pomset continues on the next line, not that nodes are omitted), in the first projection, with β a partial function from Var to Val s.t. $\beta(v) = 1$:

$$(v{\leftarrow}0) \to \{\heartsuit, \spadesuit\} \to \{\heartsuit\} \to \{\heartsuit[3/\mathsf{sw}]\} \to (v{\leftarrow}1) \to \beta \to \cdots$$

$$\begin{tikzcd}
 & & & \{\spadesuit[2/\mathsf{sw}]\} \arrow[r] & \{\spadesuit[4/\mathsf{sw}]\} \arrow[dr] & \\
\cdots \arrow[r] & \{\spadesuit\} \arrow[r] & \{\spadesuit[2/\mathsf{sw}]\} \arrow[ur] \arrow[dr] & & & \{\spadesuit[4/\mathsf{sw}], \heartsuit[4/\mathsf{sw}]\} \\
 & & & \{\heartsuit[3/\mathsf{sw}]\} \arrow[r] & \{\heartsuit[4/\mathsf{sw}]\} \arrow[ur] &
\end{tikzcd}$$

Every node labeled with a set of packets can be understood intuitively as "at this point in the execution these packets were a subset of the total packets present in the network." We can observe in the pomset that the $\heartsuit$ packet was at switch 3, before the $\spadesuit$ packet reached switch 2. We also see that $v{\leftarrow}1$, happens between $v{\leftarrow}0$ and β. In the end, both packets are observed at switch 4.

The second projection in the semantics corresponding to this pomset is the set of output packets $\{\spadesuit[4/\mathsf{sw}], \heartsuit[4/\mathsf{sw}]\}$.

In the full version of this article [38, Appendix E], we show something stronger: in all behaviors that can happen in isolation, the packet $\heartsuit[3/\mathsf{sw}]$ is recorded into the global pomset before the assignment $v{\leftarrow}1$, which precedes the observation that v equals 1 and the generation of the packet $\spadesuit[2/\mathsf{sw}]$.

We can write an axiomatic statement that captures that the above behavior is in the closed semantics of p on input $\{\heartsuit, \spadesuit\}$. To do this, we first need to capture the pictured global state pomset with corresponding set of output packets syntactically, for which we use an abbreviation. Namely, we can write a program that outputs, on any input, a specific packet: for a packet π, we write this program simply as π. The output of $[\![\pi]\!]$ on any input is $\{\mathbf{1} \cdot \{\pi\}\}$. This extends to sets of packets: $\heartsuit \parallel \spadesuit$ denotes a program whose semantics is $\{\mathbf{1} \cdot \{\heartsuit \parallel \spadesuit\}\}$ on any input. This notation pairs well with the use of the letters $a \in 2^{\mathsf{Pk}}$ as programming syntax: if we know which set of packets we (want to) record into the global state pomset with dup, we can also directly write this set of packets in the program as a syntactic letter. For instance, the program $(\heartsuit \parallel \spadesuit) \mathbin{;} \mathsf{dup}$, has the same behaviors as $(\heartsuit \parallel \spadesuit) \mathbin{;} \{\heartsuit, \spadesuit\}$: the moment we execute the dup, we know the current set of packets is $\{\heartsuit, \spadesuit\}$, and thus writing this set of packets as a letter and recording that letter into the global state pomset will have the same result. Using these two pieces of information, we can write the program

$$\begin{aligned}
q \triangleq \Big(&(v{\leftarrow}0) \mathbin{;} \{\heartsuit, \spadesuit\} \mathbin{;} \{\heartsuit\} \mathbin{;} \{\heartsuit[3/\mathsf{sw}]\} \mathbin{;} (v{\leftarrow}1) \mathbin{;} (v{=}1) \mathbin{;} \{\spadesuit\} \mathbin{;} \ldots \\
&\ldots \{\spadesuit[2/\mathsf{sw}]\} \mathbin{;} \Big((\{\spadesuit[2/\mathsf{sw}]\} \mathbin{;} \{\spadesuit[4/\mathsf{sw}]\}) \parallel (\{\heartsuit[3/\mathsf{sw}]\} \mathbin{;} \{\heartsuit[4/\mathsf{sw}]\})\Big) \mathbin{;} \ldots \\
&\ldots \{\spadesuit[4/\mathsf{sw}], \heartsuit[4/\mathsf{sw}]\}\Big) \mathbin{;} (\spadesuit[4/\mathsf{sw}] \parallel \heartsuit[4/\mathsf{sw}])
\end{aligned} \tag{1}$$

The first chunk of this program is the syntactic encoding of the desired global state pomset, where the $\heartsuit$ packet arrives at switch 3 before the $\spadesuit$ packet arrives at switch 2, and the final parallel of packets represents the set of output packets. We can prove using the axioms of $\mathsf{CNetKAT}$ that

$$(\heartsuit \parallel \spadesuit) \mathbin{;} q \leqq (\heartsuit \parallel \spadesuit) \mathbin{;} p \tag{2}$$

(2) states that the behavior of q on input $\{\heartsuit, \spadesuit\}$, is included in the behavior of p on the same input. In the behavior of q, it is clear that the $\heartsuit$ packets are observed at switch 3 before the $\spadesuit$ packets appear at switch 2.

Remark 7 (Generalized alphabet). Here we see the use of sets of packets as letters in the program syntax. Program q is much closer to the behavior we try to capture, and therefore easier to analyze, than a program containing dup.

To check the validity of equivalences such as (2), we axiomitize CNetKAT and prove it sound and complete. The axioms include the axioms of KA, extended with additional axioms for operations that manipulate packets and the global state. The full axiomatization appears in Section 3.4. For instance, $\mathsf{drop} ; q \equiv \mathsf{drop}$ states that no outputs are produced in the absence of inputs. The program drop drops the set of inputs and returns $\{\mathbf{1} \cdot \varnothing\}$. Any program q after drop outputs $\{\mathbf{1} \cdot \varnothing\}$, because q is not executed when the input is empty. In contrast, $q \,; \mathsf{drop} \equiv \mathsf{drop}$ does not hold since q might have changed the global state.

In addition to drop, CNetKAT has a program abort, which acts as a unit for non-deterministic choice ($+$). To illustrate the difference between abort and drop consider $(f{=}n) \,; (f{=}m)$ and $(v{=}n) \wedge (v{=}m)$, where $m \neq n$. The first program filters using $f = n$ and and then filters using $f = m$ where $m \neq n$. This yields $\{\mathbf{1} \cdot \varnothing\}$, since a packet cannot have different values for f. Hence, we can derive $(f{=}n) \,; (f{=}m) \equiv \mathsf{drop}$. The second program asserts the global state variable v has value n and m, which is inconsistent; we require variable v to have two different values at the same time. Hence, from the axioms we can derive that $(v{=}n) \wedge (v{=}m) \equiv \bot \equiv \mathsf{abort}$.

We prove in §4 that the axiomatization presented in Section 3.4 is not only sound but also complete—i.e., all programs with the same semantics can be proved equivalent using the axioms. The rest of the paper is devoted to presenting the CNetKAT syntax and semantics formally (§3), and establishing conservativity results over NetKAT and POCKA. Lastly we present a case study (§5).

3 Concurrent NetKAT

This section defines the syntax and semantics of CNetKAT formally.

3.1 Pomsets and pomset languages

For a poset $(X, \leq)$ and a set $S \subseteq X$, define the *downwards-closure* of S by $S_{\leq} ::= \{x \mid \exists y \in S \text{ s.t } x \leq y\}$ and $P_{\leq}(X) ::= \{Y \subseteq X \mid Y = Y_{\leq}\}$. It is well-known that $P_{\leq}(X)$ carries the structure of a bounded distributive lattice, with intersection as meet, union as join, X as top and $\varnothing$ as bottom. Further, if $(X, \leq)$ is finite, the lattice is itself finite and thus carries a (necessarily unique) pseudocomplement defined by $\overline{Y} ::= \bigcup\{Z \in P_{\leq}(X) \mid Y \cap Z = \varnothing\}$. We provide a concrete lattice with a pseudocomplement below.

Pomsets are used to capture the different evolutions of the state as it is accessed concurrently by different threads. Pomsets are labeled posets (up to isomorphism), used as a generalization of words [11,12]. A *labeled poset* over a finite alphabet Σ is a triple $\mathbf{u} = \langle S_{\mathbf{u}}, \leq_{\mathbf{u}}, \lambda_{\mathbf{u}} \rangle$, where $(S_{\mathbf{u}}, \leq_{\mathbf{u}})$ is a partially ordered set and $\lambda_{\mathbf{u}} \colon S \to \Sigma$ is the labeling function. For $\mathbf{u}, \mathbf{v}$ labeled posets, we say $\mathbf{u}$

is *isomorphic* to $\mathbf{v}$, $\mathbf{u} \cong \mathbf{v}$, if there exists a bijection $h\colon S_{\mathbf{u}} \to S_{\mathbf{v}}$ that preserves labels — $\lambda_{\mathbf{v}} \circ h = \lambda_{\mathbf{u}}$— and preserves and reflects ordering— $s \leq_{\mathbf{u}} s'$ if and only if $h(s) \leq_{\mathbf{v}} h(s')$. A *pomset* over Σ is an isomorphism class of labeled posets over Σ, i.e., the class $[\mathbf{v}] = \{\mathbf{u} \mid \mathbf{u} \cong \mathbf{v}\}$ for some labeled poset $\mathbf{v}$. Because pomsets are label-preserving isomorphism classes, the nature of the carrier is not relevant, only its cardinality and order. The triple $\mathbf{u} = \langle S_{\mathbf{u}}, \leq_{\mathbf{u}}, \lambda_{\mathbf{u}}\rangle$ is a representation of the pomset. However, often we abuse terminology and call $\mathbf{u}$ the pomset.

We write $\mathcal{P}_{\mathrm{om}}(\Sigma)$ for the set of pomsets over Σ, and $\mathbf{1}$ for the empty pomset. When $a \in \Sigma$, we write a for the pomset represented by the labeled poset with a single node labeled by a. Pomsets can be composed sequentially and in parallel.

The *parallel composition* of two pomsets is obtained by taking the disjoint union of the carriers, while keeping the ordering relations within each component. Formally, $\mathbf{u} \parallel \mathbf{v} = \left\langle S_{\mathbf{u}\parallel\mathbf{v}}, \leq_{\mathbf{u}\parallel\mathbf{v}}, \lambda_{\mathbf{u}\parallel\mathbf{v}}\right\rangle$, with $S_{\mathbf{u}\parallel\mathbf{v}} = S_{\mathbf{u}} + S_{\mathbf{v}}$, $\leq_{\mathbf{u}\parallel\mathbf{v}} = \leq_{\mathbf{u}} \cup \leq_{\mathbf{v}}$ and $\lambda_{\mathbf{u}\parallel\mathbf{v}}(x) = \lambda_{\mathbf{u}}(x)$, for $x \in S_{\mathbf{u}}$, and $\lambda_{\mathbf{u}\parallel\mathbf{v}}(x) = \lambda_{\mathbf{v}}(x)$, for $x \in S_{\mathbf{v}}$. Two pomsets are composed *sequentially* by taking the disjoint union of the carriers and ordering all elements of the first before all elements of the second, keeping the ordering relations within each component. Formally, $\mathbf{u} \cdot \mathbf{v} = \langle S_{\mathbf{u}\cdot\mathbf{v}}, \leq_{\mathbf{u}\cdot\mathbf{v}}, \lambda_{\mathbf{u}\cdot\mathbf{v}}\rangle$, with $S_{\mathbf{u}\cdot\mathbf{v}} = S_{\mathbf{u}} + S_{\mathbf{v}}$, $\leq_{\mathbf{u}\cdot\mathbf{v}} = \leq_{\mathbf{u}} \cup \leq_{\mathbf{v}} \cup (S_{\mathbf{u}} \times S_{\mathbf{v}})$ and $\lambda_{\mathbf{u}\cdot\mathbf{v}} = \lambda_{\mathbf{u}\parallel\mathbf{v}}$.

Gischer introduced a notion of ordering on pomsets [11]: $\mathbf{u} \sqsubseteq \mathbf{v}$ means that $\mathbf{u}, \mathbf{v}$ have the same events and labels, but $\mathbf{u}$ is "more sequential" than $\mathbf{v}$ in the sense that more events are ordered. Formally, $\mathbf{u} \sqsubseteq \mathbf{v}$ if there exists a label- and order-preserving bijection $h\colon S_{\mathbf{v}} \to S_{\mathbf{u}}$.

Pomset languages are simply sets of pomsets. The operations on pomsets lift pointwise to pomset languages, see Figure 3. The semantics of concurrent threads requires ensuring a closure property. In particular, we will close pomset languages under the subsumption order of Gischer. Additionally, for pomsets that contain nodes labeled by observations, we make use of a *contraction* order: $\mathbf{u} \preceq \mathbf{v}$, capturing that $\mathbf{u}$ results from $\mathbf{v}$ by eliminating consecutive observations that can be collapsed into one. As an example, consider

$$\begin{tikzcd}
& \alpha \arrow[dr] & \\
a \arrow[ur] \arrow[dr] & & c \\
& b \arrow[ur] &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
& \alpha \arrow[r] & \alpha \arrow[dr] & \\
a \arrow[ur] \arrow[drr] & & & c \\
& & b \arrow[ur] &
\end{tikzcd}$$

Denote these pomset with $\mathbf{u}$ and $\mathbf{v}$ respectively, and let $\alpha \in \mathsf{St}$. Then $\mathbf{u} \preceq \mathbf{v}$. A formal definition can be found in the full version of this article [38, Appendix A].

Definition 1 (Closure). *Let L be a pomset language.*

$$L{\downarrow}^{\mathsf{exch}} = \{\mathbf{u} \mid \exists \mathbf{v} \in L \text{ s.t. } \mathbf{u} \sqsubseteq \mathbf{v}\} \qquad L{\downarrow}^{\mathsf{contr}} = \{\mathbf{u} \mid \exists \mathbf{v} \in L \text{ s.t. } \mathbf{u} \preceq \mathbf{v}\}$$

We define $L{\downarrow}^{\mathsf{contr}\cup\mathsf{exch}}$ as the smallest language containing L and satisfying that if $\mathbf{v} \in L{\downarrow}^{\mathsf{contr}\cup\mathsf{exch}}$ and $\mathbf{u} \preceq \mathbf{v}$ or $\mathbf{u} \sqsubseteq \mathbf{v}$, then $\mathbf{u} \in L{\downarrow}^{\mathsf{contr}\cup\mathsf{exch}}$.

Closure under $\sqsubseteq$ is called exch because it ensures soundness of the *exchange law*, an axiom introduced in [13] to capture the possibility of interleaving. Closure under contraction is motivated algebraically; it ensures soundness of one of the axioms necessary when adding a test algebra (a PCDL or a BA) to a KA [16].

3.2 CNetKAT: syntax and semantics

CNetKAT expressions denote (possibly concurrent) packet processing programs that have access to a global state. Syntactically, CNetKAT is a language built from alphabets of tests and actions, each of which is divided in two categories. For packet tests, we firstly inherit NetKAT's *packet predicates*, which are elements of a Boolean algebra generated by an alphabet of basic tests on packet fields. Packet predicates t, u include constants drop and pass, denoting false and true, basic tests $f{=}n$, negation $\neg t$, disjunction $t \vee_{\mathcal{B}} u$ and conjunction $t \wedge_{\mathcal{B}} u$ operations.

Additionally, we have state observations, which do not have the structure of a Boolean algebra but instead form a pseudocomplemented distributive lattice. Intuitively, the functions denoting the state are partial. State observations o, o' include constants $\bot$ and $\top$, basic tests $v{=}n$, pseudocomplement $\overline{o}$, intersection $o \wedge o'$ and union $o \vee o'$. The other constructs were introduced in §2 (see Figure 2).

The semantics of a program is a function $[\![\cdot]\!] \colon 2^{\mathsf{Pk}} \to 2^{\mathfrak{P}\mathrm{om}(\mathsf{St} \cup \mathsf{Act} \cup 2^{\mathsf{Pk}}) \cdot 2^{\mathsf{Pk}}}$ that takes a set of packets a and produces a (possibly empty) set of pairs $\mathbf{u} {\cdot} b$ consisting of a pomset $\mathbf{u}$, recording the global state behavior and the storage of local packets whenever dup is used, and a set of packets b. On an empty input set, every program produces $\{\mathbf{1} \cdot \varnothing\}$, modeling that nothing can happen without packets. Producing the empty set when the input is non-empty models a program that has aborted, whereas producing a set $\{\mathbf{1} {\cdot} \varnothing\}$ models dropping all the packets without any change to the state. Most of the semantics was already explained in §2; in the following we elaborate on some behaviors and illustrate subtleties concerning the units. See Figure 3 for an overview of the full denotational semantics of CNetKAT.

On a non-empty input a, a packet filter t removes packets in a that do not satisfy predicate t and does not touch the state — this is captured by the set $\{\mathbf{1} \cdot [\![t]\!]_{\mathcal{B}}(a)\}$, where $[\![t]\!]_{\mathcal{B}}(a)$ is interpreted as an element of the Boolean algebra $(2^a, \cup, \cap, \varnothing, a, \setminus)$ defined by the poset $(2^a, \subseteq)$, and $[\![t]\!]_{\mathcal{B}}(a)$ is defined as the homomorphic extension of $[\![f{=}n]\!]_{\mathcal{B}}(a) = \{\pi \in a \mid \pi(f) = n\}$.

A state observation denotes a function that returns a set with elements $\mathbf{u} \cdot a$ when applied to a set a. In case the original input set a is empty, nothing happens and the output of $[\![o]\!](a)$ is simply $\{\mathbf{1} {\cdot} \varnothing\}$. When a is not empty, the semantics of o makes use of an observation algebra developed in [14,37]. More formally, we take the pseudocomplemented bounded distributive lattice $(P_{\leq}(\mathsf{St}), \cup, \cap, \mathsf{St}, \varnothing, \overline{\cdot},)$ generated by the poset $(\mathsf{St}, \leq)$ with $\alpha \leq \beta$ if and only if $\mathsf{domain}(\beta) \subseteq \mathsf{domain}(\alpha)$ and $\forall x \in \mathsf{domain}(\beta).\alpha(x) = \beta(x)$. Then, a state observation is interpreted as $\mathsf{St}^* \cdot [\![o]\!]_{\mathcal{O}} \cdot \mathsf{St}^* \times \{a\}$, where $[\![o]\!]_{\mathcal{O}}$ is an element of $P_{\leq}(\mathsf{St})$ and defined as the homomorphic extension of the assignment $[\![v{=}n]\!]_{\mathcal{O}} = \{\alpha \in \mathsf{St} \mid \alpha(v) = n\}$. Intuitively, in $[\![o]\!]_{\mathcal{O}}$, we find all the partial functions (elements of St) that agree with o. For instance, $[\![v{=}n]\!]_{\mathcal{O}}$ contains all partial functions that assign n to v. This also illustrates the need for a pseudocomplement rather than a complement: if threads have only partial information about the state, an observation should be satisfied only if there is *positive evidence* for it. Hence, e.g. $\overline{v{=}n}$ should be satisfied only if v has a value and it is not n, which is not captured by the complement from a Boolean algebra — the complement would also include partial functions that do not assign a value to v in the behavior of $\overline{v{=}n}$. This is incorrect, because if v has no value in a partial observation, we might learn later that the actual value of v was in fact n, and it was therefore incorrect to assert $\overline{v{=}n}$.

State modifications are interpreted as a set of elements $\mathbf{u}\cdot a$ when applied to a set a. The pomsets $\mathbf{u}$ record the state modification surrounded by arbitrary state observations; in the first projection of the semantics of the assignment $v{\leftarrow}n$ we get a set of possible pomsets: $\mathsf{St}^* \odot \{v \leftarrow n\} \odot \mathsf{St}^*$.

Remark 8. We surround state changes and observations with arbitrary sequences of states to include global pomsets that have alternating modifications and states in the semantics. Reasoning about behavior of programs is more practical using such alternating pomsets, because the states allow one to take stock of the configuration of the machine in between modifications. The semantics contains also non-alternating pomsets to ensure compositionality w.r.t the parallel.

CNetKAT has six different syntactical units, some of which coincide semantically. There are two units for packets: drop, which drops all the packets ($\{\mathbf{1}\cdot\varnothing\}$), and pass, which passes the current packets without changing the state ($\{\mathbf{1}\cdot a\}$ on input a). Similarly, we have two units for state observations: $\bot$ and $\top$. The first one indicates an inconsistent state, and therefore the whole program exhibits no behavior; its behavior is $\varnothing$. The second one indicates any state observation is acceptable, and its behavior on input a is $\{s \cdot a \mid s \in \mathsf{St}\}$. Lastly there are two units for programs in general: abort, the program without behavior, and skip, the program where nothing happens (on input a its semantics is $\{\mathbf{1}\cdot a\}$). Hence, abort is equivalent to $\bot$ and skip equivalent to pass. All units behave as $\{\mathbf{1}\cdot\varnothing\}$ when the input set is $\varnothing$, because nothing happens when there are no packets.

The CNetKAT semantics consists of pairs of global state pomsets and sets of output packets. It might be possible to encode the information of the output packets as a final node in the pomset, but keeping the set of output packets separated allows us to easily track the input-output behavior of a program in terms of packets. This brings CNetKAT closer to NetKAT and its packet processing behavior. In particular, the NetKAT packet processing axioms, can only be used because we track the input-output behavior of the program separately.

To obtain the full semantics, and ensure we capture correctly the intended behavior, we need to perform a closure on the state component.

Definition 2 (Closed Semantics). *Given a CNetKAT policy p, we define the semantics of p when applied to input $a \in 2^{\mathsf{Pk}}$ as*

$$[\![p]\!]\!\downarrow(a) = \left\{\mathbf{u}\cdot b \mid \mathbf{v}\cdot b \in [\![p]\!](a), \mathbf{u} \in \{\mathbf{v}\}\!\downarrow^{\mathsf{contr}\cup\mathsf{exch}}\right\}$$

Closure under exch and contr formalizes important intuitions about the semantics of concurrent threads. The closure under exch ensures all traces resulting from interleaving threads are included, and the closure under contr specifies that if two observations hold simultaneously, then it is possible to observe them in sequence. Note that the converse should not hold as some action could happen in between the two observations in a parallel thread.

We distinguish state, packet and deterministic packet programs as follows.

Definition 3 (State and deterministic packet programs). *Let $\mathcal{T}_{\mathsf{packet}}$ denote packet programs, which are programs generated by the following grammar:*

$$p, q ::= t \in \mathcal{B} \cup \{f \leftarrow n \mid f \in \mathsf{Fld}, n \in \mathsf{Val}\} \ \mid p + q \ \mid \ p \,;\, q \ \mid \ p \parallel q \ \mid \ p^*$$

Let $\mathcal{T}_{\mathsf{state}}(\Sigma)$ denote state programs over alphabet Σ:

$$s, v ::= \mathsf{abort} \ \mid \ \mathsf{skip} \ \mid \ u \in \Sigma \ \mid \ s + v \ \mid \ s \,;\, v \ \mid \ s \parallel v \ \mid \ s^*$$

Let $\mathcal{T}_{\mathsf{det-pack}}$ denote deterministic packet programs:[7]*:*

$$x, y ::= t \in \mathcal{B} \cup \{f \leftarrow n \mid f \in \mathsf{Fld}, n \in \mathsf{Val}\} \ \mid \ x \,;\, y \ \mid \ x \parallel y$$

In this paper we mostly use state programs over alphabet $\mathcal{O} \cup \mathsf{Act} \cup 2^{\mathsf{Pk}} \cup \{\mathsf{dup}\}$. Whenever we intend to use this alphabet, we simply write $\mathcal{T}_{\mathsf{state}}$.

We prove the following lemmas regarding the CNetKAT semantics.

Lemma 1 (State and packet program semantics). *Let $p \in \mathcal{T}_{\mathsf{packet}}$, $s \in \mathcal{T}_{\mathsf{state}}$ and $a \in 2^{\mathsf{Pk}}$. For all $w \in [\![p]\!](a)$, w is of the form $\mathbf{1} \cdot b$ for $b \in 2^{\mathsf{Pk}}$. For all $w \in [\![s]\!](a)$, w is of the form $\mathbf{v} \cdot a$ for $\mathbf{v}$ a pomset over $\mathsf{St} \cup \mathsf{Act} \cup 2^{\mathsf{Pk}}$.*

For non-empty sets of packets a and a', the global behavior of a state program without dup is identical on both inputs. Let $2^{\mathsf{Pk}}_{\mathsf{ne}}$ denote $2^{\mathsf{Pk}} \setminus \{\varnothing\}$.

Lemma 2. *Let $s \in \mathcal{T}_{\mathsf{state}}(\mathcal{O} \cup \mathsf{Act} \cup 2^{\mathsf{Pk}})$. For all $a, a' \in 2^{\mathsf{Pk}}_{\mathsf{ne}}$ we have $\{\mathbf{u} \mid \mathbf{u} \cdot b \in [\![s]\!](a)\} = \{\mathbf{u} \mid \mathbf{u} \cdot b \in [\![s]\!](a')\}$.*

We characterize $[\![-]\!]_{\mathcal{B}}$ in terms of its behavior on subsets of the input set.

Lemma 3. *Let $t \in \mathcal{B}$ and $a, b \subseteq \mathsf{Pk}$. Then $[\![t]\!]_{\mathcal{B}}(a \cup b) = [\![t]\!]_{\mathcal{B}}(a) \cup [\![t]\!]_{\mathcal{B}}(b)$.*

Lastly, we have a lemma characterising the semantics of a deterministic packet program in terms of its behavior on subsets of the input.

Lemma 4. *Let $x \in \mathcal{T}_{\mathsf{det-pack}}$ and $a, b \subseteq \mathsf{Pk}$. Then*
$[\![x]\!](a \cup b) = \{\mathbf{1} \cdot (c \cup d) \mid [\![x]\!](a) = \{\mathbf{1} \cdot c\}, [\![x]\!](b) = \{\mathbf{1} \cdot d\}\}$.

3.3 Is CNetKAT conservative over NetKAT and POCKA?

CNetKAT combines NetKAT and POCKA, so it is natural to ask whether it is a conservative extension of either language. It turns out that the answer is positive for POCKA, and for a fragment of NetKAT. We start by recalling the semantics of NetKAT [3]. Note that NetKAT expressions are packet programs without $\parallel$.

[7] Equivalently, we can define $\mathcal{T}_{\mathsf{packet}}$ by adding a predicate H to the signature of our algebra that counts the number of $*$'s and $+$'s a term contains, and a packet program p is an element of $\mathcal{T}_{\mathsf{det-pack}}$ if and only if $p \in \mathcal{T}_{\mathsf{packet}}$ and $H(p) = 0$.

Definition 4 (NetKAT **semantics).** *Let $\pi \in \mathsf{Pk}$, $t \in \mathcal{B}$ and p, q* NetKAT *terms.*

$$[\![t]\!]_{\mathsf{NK}}(\pi) = [\![t]\!]_{\mathcal{B}}(\{\pi\}) \qquad [\![\mathsf{pass}]\!]_{\mathsf{NK}}(\pi) = \{\pi\} \qquad [\![\mathsf{drop}]\!]_{\mathsf{NK}}(\pi) = \{\}$$

$$[\![f \leftarrow n]\!]_{\mathsf{NK}}(\pi) = \{\pi[n/f]\} \qquad [\![p\,;q]\!]_{\mathsf{NK}}(\pi) = \bigcup\nolimits_{\pi' \in [\![p]\!](\pi)} [\![q]\!]_{\mathsf{NK}}(\pi')$$

$$[\![p^*]\!]_{\mathsf{NK}}(\pi) = \bigcup\nolimits_n [\![p^n]\!]_{\mathsf{NK}}(\pi) \qquad [\![p+q]\!]_{\mathsf{NK}}(\pi) = [\![p]\!]_{\mathsf{NK}}(\pi) \cup [\![q]\!]_{\mathsf{NK}}(\pi)$$

Theorem 1. *Take $\pi \in \mathsf{Pk}$ and NetKAT term p. $[\![p]\!]_{\mathsf{NK}}(\pi) = \bigcup_{\mathbf{1}\cdot a' \in [\![p]\!](\{\pi\})} a'$.*

We can derive a further relation between the semantics if we assume there is no use of $+$ and $*$ (the proof uses Lemma 3).

Lemma 5. *Let p be built out of packet predicates and modifications ($f \leftarrow n$), and their sequential composition. Then $[\![p]\!](a) = \left\{\mathbf{1} \cdot \bigcup_{x \in a} [\![p]\!]_{\mathsf{NK}}(x)\right\}$.*

It is worth remarking that the equational theories of NetKAT and CNetKAT are not equivalent: there are equivalent programs in NetKAT, that cannot be proved equivalent with the CNetKAT axioms, as the following example illustrates. Consider the program $p + \mathsf{drop}$ for p a packet program without parallel. In NetKAT, because the $+$ is interpreted as multicast, this program is provably equivalent to p: executing p on your input packet while at the same time also dropping a copy of the input, has the same outcome as just executing p. In CNetKAT, however, this is not the case. Instead, the $+$-operator is interpreted as non-deterministic choice and in the semantics of $p + \mathsf{drop}$ we get the trace $\mathbf{1} \cdot \varnothing$, representing the choice of dropping all the packets, which is not present in the semantics of p. Hence, this axiom is unsound ($p + \mathsf{drop} \not\equiv p$), and instead the alternative axiom $p \parallel \mathsf{drop} = p$ holds, reflecting the fact that $\parallel$ is multicast.

We now show CNetKAT semantics is equivalent to the POCKA semantics on state programs. In [37], POCKA terms are what we defined as state programs over the alphabet $\mathcal{O} \cup \mathsf{Act}$, and they are interpreted in terms of pomset languages over assignments and states, encoded as partial functions, similarly to separation logic [33]. The POCKA semantics are defined in two steps: the first step results in a set containing all pomsets that can be derived directly from the terms, and in a second step this set is closed under two laws—exch and contr—that account for all traces that can be built in parallel threads (including simple interleaving).

Definition 5 (POCKA **semantics).** *Let $o \in \mathcal{O}$, $e \in \mathsf{Act}$, $p, q \in \mathcal{T}_{\mathsf{state}}(\mathcal{O} \cup \mathsf{Act})$.*

$$\begin{array}{llll} (\!|o|\!) = \mathsf{St}^* \odot [\![o]\!]_{\mathcal{O}} \odot \mathsf{St}^* & (\!|p\,;q|\!) = (\!|p|\!) \odot (\!|q|\!) & (\!|\mathsf{skip}|\!) = \{\mathbf{1}\} & (\!|\mathsf{abort}|\!) = \varnothing \\ (\!|e|\!) = \mathsf{St}^* \odot \{e\} \odot \mathsf{St}^* & (\!|p \parallel q|\!) = (\!|p|\!) \parallel (\!|q|\!) & (\!|p^*|\!) = (\!|p|\!)^* & (\!|p+q|\!) = (\!|p|\!) \cup (\!|q|\!) \end{array}$$

The semantics of a POCKA *expression p is $[\![p]\!]_{\mathsf{POCKA}} = (\!|p|\!){\downarrow}^{\mathsf{exch} \cup \mathsf{contr}}$.*

Theorem 2. *CNetKAT is a conservative extension of* POCKA*: if p is a* POCKA *term ($p \in \mathcal{T}_{\mathsf{state}}(\mathcal{O} \cup \mathsf{Act})$) then for $a \neq \varnothing$, $[\![p]\!]{\downarrow}(a) = \{\mathbf{u} \cdot a \mid \mathbf{u} \in [\![p]\!]_{\mathsf{POCKA}}\}$.*

3.4 Axiomatization

We introduce notation to describe packets and sets of packets axiomatically. Let $f_1, \ldots, f_k$ be a list of all fields of a packet in some fixed order. Then for each tuple $\overline{n} = n_1, \ldots, n_k$ we obtain expressions $f_1 = n_1 \cdots f_k = n_k$ and $f_1 \leftarrow n_1 \cdots f_k \leftarrow n_k$, which, similar to NetKAT, we call *complete tests* and *complete assignments*. Complete tests are also referred to as atoms, because they are the atoms of the Boolean algebra generated by the tests. We denote the set of atoms by At, complete tests with α and complete assignments with π. There is a one-to-one correspondence between complete tests and assignments according to the values of $\overline{n}$. For $\alpha \in \mathsf{At}$ we denote the corresponding complete assignment by π_α, and if π is a complete assignment we denote the corresponding atom by α_π.

There is also a link between sets of packets and terms of the form $\|_{i \in I} \pi_i$. For each set of packets a, we take the set $\{\pi_i \mid i \in I\}$ of complete assignments such that each π_i corresponds to a packet of a, and combine them in parallel. Formally, for a set of packets a there exists an expression $\|_{i \in I} \pi_i$, that we denote with Π_a, such that on any input $b \neq \varnothing$, $[\![\Pi_a]\!](b) = \{\mathbf{1} \cdot a\}$. Similarly, the semantics of an expression of the form $\|_{i \in I} \pi_i$ on any input is always $\{\mathbf{1} \cdot a\}$ for some $a \in 2^{\mathsf{Pk}}$. We use the notation Π_a as a syntactic representation of set of packets a.

$\mathsf{CNetKAT}$ has the structure of a Kleene algebra on state programs, enriched with additional axioms. Tests form a Boolean algebra and state observations a pseudocomplemented distributive lattice (PCDL). The test and observation structures are subject to interaction constraints. The packet processing behavior is captured by the packet axioms, which contain axioms for individual packets and sets of packets. The axioms governing the parallel operator are partially familiar from earlier work on BKA [13,25]. There is also the exchange law familiar from CKA. Lastly, we have axioms for the interactions between state programs and packet programs. The full set of axioms is described in Figure 4. We write $\equiv$ for the smallest congruence on Prg generated by the axioms in Figure 4.

Remark 9 (When is Π_a equal to drop*?).* $\Pi_a \equiv \mathsf{drop}$ if and only if a is empty. $\Pi_\varnothing = \|_{i \in \varnothing} \pi_i \equiv \|\varnothing \equiv \bigvee \varnothing \equiv \mathsf{drop}$. For all other a, we have $\Pi_a \not\equiv \mathsf{drop}$.

There are a few subtleties to notice in Figure 4. First, we point out the interaction between drop and abort. When no packets are present, not even abort can be executed. Hence, if we drop all packets and then abort, the abort does not happen: $\mathsf{drop} \,;\, \mathsf{abort} \equiv \mathsf{drop}$. On the other hand, if we first abort and then drop all the packets, the behavior is equal to just aborting: $\mathsf{abort} \,;\, \mathsf{drop} \equiv \mathsf{abort}$.

In the axioms of the parallel operator, the axiom $s \parallel \mathsf{skip} \equiv \mathsf{skip}$ from BKA is missing; it only holds when s is a state program, and can be found in the local state vs global state axioms. In addition to the familiar BKA axioms, there is the axiom $\mathsf{drop} \parallel p \equiv p$, in contrast with $\mathsf{abort} \parallel p \equiv \mathsf{abort}$.

The local state vs global state axioms capture the interactions between the global pomset and the output packets. The first one, $\Pi_a \,;\, \mathsf{dup} \equiv \Pi_a \,;\, a$, captures the intuition that if we know the input is a (due to Π_a, which, as a parallel of complete assignments, essentially overwrites any non-empty input set to a), then we know the dup is recording an "a". The second axiom, $\Pi_a \,;\, w \equiv w \,;\, \Pi_a$ states

Kleene Algebra axioms $s \in \mathcal{T}_{\mathsf{state}}$

$$\begin{aligned}
p + (q + r) &\equiv (p + q) + r \\
p + q &\equiv q + p \\
p + \mathsf{abort} &\equiv p \\
p + p &\equiv p \\
p\,;(q\,;r) &\equiv (p\,;q)\,;r \\
s\,;\mathsf{abort} &\equiv \mathsf{abort} \\
\mathsf{abort}\,;p &\equiv \mathsf{abort} \\
p\,;\mathsf{skip} &\equiv p \equiv \mathsf{skip}\,;p \\
p\,;(q + r) &\equiv p\,;q + p\,;r \\
(p + q)\,;r &\equiv p\,;r + q\,;r \\
p^* &\equiv \mathsf{skip} + pp^* \\
p + q\,;r \leqq q &\Rightarrow p \cdot r^* \leqq q \\
p^* &\equiv \mathsf{skip} + p^*p \\
p + q \cdot r \leqq r &\Rightarrow q^* \cdot p \leqq r
\end{aligned}$$

Packet axioms $x \in \mathcal{T}_{\mathsf{det-pack}}$

$$\begin{aligned}
&f{=}n\,;f'{\leftarrow}m \equiv f'{\leftarrow}m\,;f{=}n \quad (f \neq f') \\
&f{\leftarrow}n\,;f'{\leftarrow}m \equiv f'{\leftarrow}m\,;f{\leftarrow}n \quad (f \neq f') \\
&f{=}n\,;f{\leftarrow}n \equiv f{=}n \\
&f{\leftarrow}n\,;f{=}n \equiv f{\leftarrow}n \\
&f{\leftarrow}m\,;f{\leftarrow}n \equiv f{\leftarrow}n \\
&x \parallel x \equiv x \\
&x\,;(p \parallel q) \equiv (x\,;p) \parallel (x\,;q) \\
&(p \parallel q)\,;x \equiv (p\,;x) \parallel (q\,;x)
\end{aligned}$$

Local vs global state $y, z \in \mathcal{T}_{\mathsf{packet}}$, $s, v \in \mathcal{T}_{\mathsf{state}}$, $w \in \mathcal{T}_{\mathsf{state}}(\mathcal{O} \cup \mathsf{Act} \cup 2^{\mathsf{Pk}})$

$$\begin{aligned}
&\Pi_a\,;\mathsf{dup} \equiv \Pi_a\,;a \quad (a \in 2^{\mathsf{Pk}}) \\
&\Pi_a\,;w \equiv w\,;\Pi_a \quad (a \in 2^{\mathsf{Pk}}_{\mathsf{ne}}) \\
&\mathsf{drop}\,;p \equiv \mathsf{drop} \qquad y\,;\mathsf{drop} \equiv \mathsf{drop} \\
&s \parallel \mathsf{skip} \equiv s \\
&(s\,;y) \parallel (v\,;z) \equiv (s \parallel v)\,;(y \parallel z)
\end{aligned}$$

Extensionality

$$\forall a \in 2^{\mathsf{Pk}}.(\Pi_a\,;p \equiv \Pi_a\,;q) \Rightarrow p \equiv q$$

Parallel axioms

$$\begin{aligned}
&p \parallel (q \parallel r) \equiv (p \parallel q) \parallel r \\
&p \parallel \mathsf{abort} \equiv \mathsf{abort} \\
&\mathsf{drop} \parallel p \equiv p \\
&p \parallel (q + r) \equiv p \parallel q + p \parallel r \\
&p \parallel q \equiv q \parallel p
\end{aligned}$$

Exchange law $s, s', v, v' \in \mathcal{T}_{\mathsf{state}}$

$$(s \parallel s')\,;(v \parallel v') \leqq (s\,;v) \parallel (s'\,;v')$$

Packet pred., state obs. axioms

$\vee \in \{\vee, \vee_{\mathcal{B}}\}$, $\wedge \in \{\wedge, \wedge_{\mathcal{B}}\}$, $a, b, c \in \mathcal{B} \cup \mathcal{O}$

$$\begin{aligned}
&a \wedge b \equiv b \wedge a \\
&a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c \\
&a \vee (a \wedge b) \equiv a \equiv a \wedge (a \vee b) \\
&a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c) \\
&a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)
\end{aligned}$$

Additional state obs. axioms

$$\begin{aligned}
&o \equiv o \wedge \top \\
&o \leqq \overline{o'} \Leftrightarrow o \wedge o' \equiv \bot \\
&v = n \ \wedge\ v = m \equiv \bot \quad (n \neq m) \\
&\overline{v = n} \leqq \bigvee_{n \neq m} v = m \\
&\overline{\textstyle\bigwedge_i v_i = n_i} \leqq \textstyle\bigvee_i \overline{v_i = n_i} \quad (i \neq j.v_i \neq v_j)
\end{aligned}$$

Additional packet pred. axioms

$$\begin{aligned}
&t \vee_{\mathcal{B}} \mathsf{pass} \equiv \mathsf{pass} \equiv t \vee_{\mathcal{B}} \neg t \\
&t \wedge_{\mathcal{B}} \neg t \equiv \mathsf{drop} \\
&f{=}n \wedge_{\mathcal{B}} f{=}m \equiv \mathsf{drop} \quad (n \neq m) \\
&\textstyle\bigvee_i f{=}i \equiv \mathsf{pass}
\end{aligned}$$

Interface axioms

$$\begin{aligned}
&o \wedge o' \leqq o\,;o' \quad && o \vee o' \equiv o + o' \quad && (o, o' \in \mathcal{O}) \\
&\mathsf{abort} \equiv \bot && \mathsf{skip} \equiv \mathsf{pass} && (e \in \mathsf{Act}) \\
&\top\,;o \leqq o && o\,;\top \leqq o && (t, t' \in \mathcal{B}) \\
&\top\,;e \leqq e && e\,;\top \leqq e && \\
&t \wedge_{\mathcal{B}} t' \equiv t\,;t' && t \vee_{\mathcal{B}} t' \equiv t \parallel t' &&
\end{aligned}$$

Fig. 4: Axioms of CNetKAT. The left column contains the KA axioms, the packet axioms, the axioms for the interaction between the local and global state, and an extensionality axiom. The right column axiomatizes the $\parallel$, the algebra of packet tests (which is a Boolean algebra), and the algebra of partial state observations (which is a PCDL). The interface axioms connect both the lattice operators to the Kleene algebra ones. We write $e \leqq f$ as a shorthand for $e + f \equiv f$.

that for dup-free state program w, we can flip the order between changing the set of output packets or performing the state changes in w, as long as Π_a is not the parallel representing the empty set. This latter condition is crucial: if $a = \varnothing$, then $\Pi_a \equiv \mathsf{drop}$, and $\mathsf{drop} \,; w \equiv \mathsf{drop}$ (the global state changes in w do not get executed if we have no packets).

The axiom $\mathsf{drop} \,; p \equiv \mathsf{drop}$ for any program p captures the intuition that if there are no packets, nothing happens anymore. The other way around, $y ; \mathsf{drop} \equiv \mathsf{drop}$ is only true for y a packet program; if it was a state program, the global state changes get executed if we start with a non-empty set of input packets, making the behavior of $y \,; \mathsf{drop}$ not equivalent to drop.

Lastly, extensionality says that if two programs are equivalent on all inputs (i.e., $a \in 2^{\mathsf{Pk}}$), then the programs are equivalent. It is not clear whether this axiom is derivable from the others; we hope to settle this question in the future.

4 Soundness and Completeness

In this section we prove soundness and completeness of the CNetKAT semantics w.r.t. the axiomatization from Figure 4. For soundness, we prove that if programs p and q are provably equivalent using the axioms, they have the same semantics:

Theorem 3 (Soundness). *For all $p, q \in \mathsf{Prg}$, if $p \equiv q$, then $[\![p]\!]\!\downarrow \; = [\![q]\!]\!\downarrow$.*

Conversely, we will prove that if p and q have the same semantics on all inputs a, then $p \equiv q$. We structure the ***completeness*** proof in four parts:

1. Define a normal form for CNetKAT programs, and show that for every input set a, every program is provably equivalent to a program in normal form in which a is incorporated. In other words, the normal form of a program is dependent on the input. Similar to NetKAT, normal form programs are CNetKAT expressions over *complete assignments*. We show that we have a simplified set of axioms on complete assignments and tests.
2. Obtain completeness for Π_a-shaped programs from NetKAT completeness.
3. Using completeness of POCKA, obtain completeness for programs of the form $s \,; \Pi_a$ (and sums thereof), where s is a state program.
4. Lastly, we combine these results to prove that if p and q have the same behavior on input a, the program $\Pi_a \,; p$ is provably equivalent to $\Pi_a \,; q$.

Step 1: Normal form We prove that for every $a \in 2^{\mathsf{Pk}}$, we can write any program p as Π_a followed by a sum of state programs followed by a parallel of complete assignments. This is the most difficult step in the completeness proof.

We derive a few equivalences from Figure 4 regarding complete tests and assignments that make the proof of the normal form easier. We refer to these axioms as the *reduced* axioms. For α and β complete tests such that $\alpha \neq \beta$, π and π' complete assignments, and $a \in 2^{\mathsf{Pk}}_{\mathsf{ne}}, b \in 2^{\mathsf{Pk}}$, we can derive:

$$\pi \equiv \pi \,; \alpha_\pi \qquad \alpha \equiv \alpha \,; \pi_\alpha \qquad \pi \,; \pi' \equiv \pi' \qquad \alpha \,; \beta \equiv \mathsf{drop} \qquad \Pi_a \,; \Pi_b \equiv \Pi_b$$

All of these equivalences are easy consequences of the packet axioms, the packet predicate axioms, the axiom $t \wedge_{\mathcal{B}} t' \equiv t \,; t'$ and the fact that for all packet programs p we have $p \,; \mathsf{drop} \equiv \mathsf{drop} \equiv \mathsf{drop} \,; p$ [3]. The last reduced axiom is derived in the full version of this article [38, Lemma 14].

Theorem 4 (Normal form). *Let $p \in \mathsf{Prg}$ and $a \in 2^{\mathsf{Pk}}$. There exists a finite set J, and elements $u_j \in \mathcal{T}_{\mathsf{state}}(\mathcal{O} \cup \mathsf{Act} \cup 2^{\mathsf{Pk}})$ and $b_j \in 2^{\mathsf{Pk}}$ for each $j \in J$ s.t.*

$$\Pi_a \mathbin{;} p \equiv \Pi_a \mathbin{;} \sum_{j\in J} \left(u_j \mathbin{;} \Pi_{b_j}\right)$$

Sketch. The proof proceeds by induction on the structure of p. For instance, for an assignment $f \leftarrow n$, where we take $\Pi_a = \|_{k\in K} \pi_k$ for some non-empty finite index set K and complete assignments π_k, we derive

$$\begin{aligned}
\Pi_a \mathbin{;} f \leftarrow n &\equiv \Pi_a \mathbin{;} \Pi_a \mathbin{;} f \leftarrow n && (\Pi_a \mathbin{;} \Pi_b \equiv \Pi_b)\\
&= \Pi_a \mathbin{;} (\mathop{\|}_{k\in K} \pi_k) \mathbin{;} f \leftarrow n \\
&\equiv \Pi_a \mathbin{;} \mathop{\|}_{k\in K} (\pi_k \mathbin{;} f \leftarrow n) && ((p \parallel q) \mathbin{;} x \equiv (p \mathbin{;} x) \parallel (q \mathbin{;} x))\\
&\equiv \Pi_a \mathbin{;} \mathsf{skip} \mathbin{;} \mathop{\|}_{k\in K} \pi'_k && (p \mathbin{;} \mathsf{skip} \equiv p)
\end{aligned}$$

where π'_k is π_k with the assignment for f replaced by $f \leftarrow n$. If $K = \varnothing$ then $\Pi_a \equiv \mathsf{drop}$ and the equivalence above follows immediately. The most difficult case is the star; we use an argument that relies on the fact that matrices over a Kleene algebra form a Kleene algebra [20]. A proof can be found in the full version of this article [38, Appendix D] $\square$

Step 2: Completeness for Π_a-shaped programs As mentioned, Π_a-shaped programs are syntactic representations of packet sets. We prove that if two such programs result in the same set of packets on any non-empty input, they are provably equivalent, using that Π_a describes a unique set of packets.

Lemma 6. *Let $a \in 2^{\mathsf{Pk}}_{\mathsf{ne}}$, and $b, c \in 2^{\mathsf{Pk}}$. If $[\![\Pi_b]\!]{\downarrow}(a) = [\![\Pi_c]\!]{\downarrow}(a)$ then $\Pi_b \equiv \Pi_c$.*

Step 3: Completeness of sums in the normal form We first prove completeness for state programs, where we use completeness of POCKA. To do so, some caution is needed; POCKA terms are state terms over the alphabet $\mathcal{O} \cup \mathsf{Act}$. However, the state terms relevant here also include elements $a \in 2^{\mathsf{Pk}}$.

Lemma 7. *Let $s, v \in \mathcal{T}_{\mathsf{state}}(\mathcal{O} \cup \mathsf{Act} \cup 2^{\mathsf{Pk}})$ and $a \in 2^{\mathsf{Pk}}_{\mathsf{ne}}$. If $[\![s]\!]{\downarrow}(a) = [\![v]\!]{\downarrow}(a)$, then $s \equiv v$.*

Next we prove completeness for expressions of the form $s \mathbin{;} \Pi_a$, and then extend this to arbitrary finite sums of such programs:

Lemma 8. *Let $b, c \in 2^{\mathsf{Pk}}$, u, v state programs, and $a \in 2^{\mathsf{Pk}}_{\mathsf{ne}}$. Then we have: $[\![u \mathbin{;} \Pi_b]\!]{\downarrow}(a) = [\![v \mathbin{;} \Pi_c]\!]{\downarrow}(a) \Rightarrow u \mathbin{;} \Pi_b \equiv v \mathbin{;} \Pi_c$.*

Lemma 9. *If $\left[\!\!\left[\sum_{j\in J}(u_j \mathbin{;} \Pi_{b_j})\right]\!\!\right]{\downarrow}(a) = [\![\sum_{k\in K}(v_k \mathbin{;} \Pi_{c_k})]\!]{\downarrow}(a)$ for some $a \in 2^{\mathsf{Pk}}_{\mathsf{ne}}$, then $\sum_{j\in J}(u_j \mathbin{;} \Pi_{b_j}) \equiv \sum_{k\in K}(v_k \mathbin{;} \Pi_{c_k})$, where J, K are finite; u_j, v_k are state programs and $b_j, c_k \in 2^{\mathsf{Pk}}$ for each j, k.*

Step 4: Completeness The last lemma before proving completeness relates the semantics of p on input a to the semantics of $\Pi_a ; p$ on any non-empty input.

Lemma 10. *Let $b \in 2^{\mathsf{Pk}}_{\mathsf{ne}}$, $a \in 2^{\mathsf{Pk}}$. For all $p \in \mathsf{Prg}$, $[\![\Pi_a ; p]\!]\!\downarrow(b) = [\![p]\!]\!\downarrow(a)$.*

Theorem 5 (Completeness). *Let $p, q \in \mathsf{Prg}$. For all $a \in 2^{\mathsf{Pk}}$ we have that if $[\![p]\!]\!\downarrow(a) = [\![q]\!]\!\downarrow(a)$, then $p \equiv q$.*

Proof. We first show that $\Pi_a ; p \equiv \Pi_a ; q$ for all $a \in 2^{\mathsf{Pk}}$. In case $a = \varnothing$, Π_a must be the empty parallel. Hence, $\Pi_a ; p \equiv \mathsf{drop} \equiv \Pi_a ; q$. In the rest of the proof we assume $a \neq \varnothing$. Via Lemma 10, we obtain that $[\![p]\!]\!\downarrow(a) = [\![\Pi_a ; p]\!]\!\downarrow(a) = [\![\Pi_a ; q]\!]\!\downarrow(a) = [\![q]\!]\!\downarrow(a)$. We obtain a normal form such that $\Pi_a ; p \equiv \Pi_a ; \sum_{j \in J}(u_j ; \Pi_{b_j})$ (Theorem 4). Similarly, $\Pi_a ; q \equiv \Pi_a ; \sum_{k \in K}(v_k ; \Pi_{c_k})$. Via soundness we derive $\left[\!\!\left[\Pi_a ; \sum_{j \in J}(u_j ; \Pi_{b_j})\right]\!\!\right]\!\downarrow(a) = [\![\Pi_a ; \sum_{k \in K}(v_k ; \Pi_{c_k})]\!]\!\downarrow(a)$, and via Lemma 10 that $\left[\!\!\left[\sum_{j \in J}(u_j ; \Pi_{b_j})\right]\!\!\right]\!\downarrow(a) = [\![\sum_{k \in K}(v_k ; \Pi_{c_k})]\!]\!\downarrow(a)$. With the partial completeness result from Lemma 9, we obtain that $\sum_{j \in J}\left(u_j ; \Pi_{b_j}\right) \equiv \sum_{k \in K}(v_k ; \Pi_{c_k})$. This leads to

$$\Pi_a ; p \equiv \Pi_a ; \sum_{j \in J}\left(u_j ; \Pi_{b_j}\right) \equiv \Pi_a ; \sum_{k \in K}(v_k ; \Pi_{c_k}) \equiv \Pi_a ; q$$

Hence, we have derived that $\Pi_a ; p \equiv \Pi_a ; q$ for all $a \in 2^{\mathsf{Pk}}$. With the extensionality axiom we can conclude that $p \equiv q$. $\square$

5 Examples

This section shows how we can use CNetKAT to model and analyze several concurrent programs. We start by analyzing the running example from §2, and then proceed to a more involved example that combines the behavior of a stateful firewall, a load balancer, and an in-network cache.

5.1 Running Example

Consider again the running example from §2. Because we are ultimately interested in the behavior of the program when the packets have reached their final destination, switch 4, we will add a test sw=4 at the end of the program:

$$p \triangleq (v{\leftarrow}0) ; (p_1 \parallel p_2 \parallel p_3 \parallel p_4)^* ; (\mathsf{sw}{=}4)$$

Recall that the CNetKAT semantics of a program contains traces that are only required to model executions where the program is composed in parallel with another program, to ensure a compositional semantics for the language. However, to analyze the behavior of a program in isolation, we want to eliminate these extra traces. To do this, we follow the same strategy used in [37], where so-called *guarded* pomsets were proposed. Guarded pomsets are a subclass of pomsets that captures the characteristics of behaviors of (concurrent) programs running in isolation. For example, in a guarded pomset, if one assertion, say $v{=}0$, occurs before another assertion, say $v{=}1$, there must be an assignment $v{\leftarrow}1$ between the two asserts to account for the change. That is, in an isolated execution every change to variables must be explained by an action in the program.

To illustrate the difference between pomsets and guarded pomsets, consider our example. We unfold the Kleene star twice and evaluate the resulting program; we obtain a pair with output $\{\spadesuit[4/\mathsf{sw}], \heartsuit[4/\mathsf{sw}]\}$ and corresponding pomset,

$$\begin{tikzcd}
 & & \beta \arrow[r] & \{\spadesuit\} \arrow[r] & \spadesuit[2/\mathsf{sw}] \arrow[r] \arrow[ddr] & \{\spadesuit[2/\mathsf{sw}]\} \arrow[r] & \{\spadesuit[4/\mathsf{sw}]\} \\
(v{\leftarrow}0) \arrow[r] & \{\heartsuit, \spadesuit\} \arrow[ur] \arrow[dr] & & & & & \\
 & & \{\heartsuit\} \arrow[r] & \{\heartsuit[3/\mathsf{sw}]\} \arrow[r] & (v{\leftarrow}1) \arrow[r] \arrow[uur] & \{\heartsuit[3/\mathsf{sw}]\} \arrow[r] & \{\heartsuit[4/\mathsf{sw}]\}
\end{tikzcd}$$

where $\beta(v) = 1$. This pomset is unguarded: $\beta(v) = 1$ occurs without a cause.

The semantics also contains a pair with $\{\spadesuit[4/\mathsf{sw}], \heartsuit[4/\mathsf{sw}]\}$ and pomset,

$$\begin{tikzcd}
 & & & & \beta \arrow[r] & \{\spadesuit\} \arrow[r] & \spadesuit[2/\mathsf{sw}] \arrow[r] \arrow[ddr] & \{\spadesuit[2/\mathsf{sw}]\} \arrow[r] & \{\spadesuit[4/\mathsf{sw}]\} \\
\gamma \arrow[r] & (v{\leftarrow}0) \arrow[r] & \alpha \arrow[r] & \{\heartsuit, \spadesuit\} \arrow[ur] \arrow[dr] & & & & & \\
 & & & & \{\heartsuit\} \arrow[r] & \{\heartsuit[3/\mathsf{sw}]\} \arrow[r] & (v{\leftarrow}1) \arrow[r] \arrow[uull] \arrow[uur] & \{\heartsuit[3/\mathsf{sw}]\} \arrow[r] & \{\heartsuit[4/\mathsf{sw}]\}
\end{tikzcd}$$

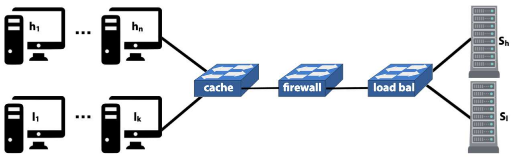
with $\alpha(v) = 0$, $\beta(v) = 1$, and γ unrestricted. This pomset is guarded because it contains an arrow from $v{\leftarrow}1$ to β, justifying the change in valuation from α to β. As we show in the full version of this article [38, Appendix E], all guarded pomsets in the semantics will have this arrow, and satisfy the desired property: $\heartsuit$ packets are observed at switch 3 before $\spadesuit$ packets are observed at switch 2.

Now consider the axiomatic claim we made in §2 (i.e., (2)), $(\heartsuit \parallel \spadesuit);q \leqq (\heartsuit \parallel \spadesuit);p$ where q is the program from (1). We can easily see that the following holds: $[\![q]\!]\!\downarrow \{\heartsuit, \spadesuit\} \subseteq [\![p]\!]\!\downarrow \{\heartsuit, \spadesuit\}$. Hence, we can use Lemma 10 and the completeness result for $\mathsf{CNetKAT}$ (Theorem 5) to obtain (2).

5.2 Stateful Load Balancer, Cache, and Firewall

For a more complex example, consider the network in Figure 5, which is adapted from an example from [2]. The overall goal is to (i) prevent packets from a high-priority server S_h going to low priority hosts $l_1, \ldots, l_k$ and (ii) load balance requests to the servers in a round robin fashion. We provide naive specifications for the cache, firewall and load balancer programs in Figure 5. For simplicity, we assume that there is exactly one low-priority host, and exactly one high-priority host, i.e., $n = k = 1$, and we leave the specification of the topology implicit.

Remark 10. In contrast with the previous example, the program in Figure 5 includes reads and writes of a global variable that occur on different physical devices. In principle, synchronizing variables like r would give rise to additional packets that update local copies of variables—a process that could itself be modelled in $\mathsf{CNetKAT}$. We leave the implementation of a translation pass that achieves the synchronization of global variables across switches to future work.



$$C \triangleq ((v{=}1) ; (\mathsf{dst}{\leftarrow}h_1) ; \mathsf{dup} + (v{=}0) ; (\mathsf{dst}{\leftarrow}l_1) ; \mathsf{dup}) \\ \parallel (\mathsf{src} = l_1 ; (\mathsf{dst}{\leftarrow}\mathsf{firewall})) \parallel (\mathsf{src} = h_1 ; (\mathsf{dst}{\leftarrow}\mathsf{firewall}))$$

$$F \triangleq (\mathsf{src} = s_h ; (v{\leftarrow}0) ; (\mathsf{dst}{\leftarrow}\mathsf{cache})) \parallel (\mathsf{src} = s_l ; (v{\leftarrow}1) ; (\mathsf{dst}{\leftarrow}\mathsf{cache})) \\ \parallel (\mathsf{src} = l_1 ; (r{\leftarrow}0) ; (\mathsf{dst}{\leftarrow}\mathsf{loadb})) \parallel (\mathsf{src} = h_1 ; (r{\leftarrow}1) ; (\mathsf{dst}{\leftarrow}\mathsf{loadb}))$$

$$L \triangleq ((r{=}1) ; (\mathsf{dst}{\leftarrow}s_h) ; \mathsf{dup} + (r{=}0) ; (\mathsf{dst}{\leftarrow}s_l) ; \mathsf{dup}) \\ \parallel \mathsf{src} = s_h ; (\mathsf{dst}{\leftarrow}\mathsf{firewall}) \parallel \mathsf{src} = s_l ; (\mathsf{dst}{\leftarrow}\mathsf{firewall})$$

Fig. 5: Stateful firewall between high/low priority hosts and servers.

In [2], the authors point out a problem with the example that arises because the cache has no means to enforce the security policy. One strategy for resolving this problem is to swap the placement of the firewall and the cache. Another is to distribute access control rules onto the cache as well as the firewall. However, there is also a second, more subtle issue: the load balancer uses the global variable r to decide to which server to forward requests. In the presence of multiple packets, another packet may arrive before the change to the global variable occurs allowing two (or more!) packets to be sent to the same server.

The issue with the load balancer can be observed in the following example. Take as input packets ♠ and ♡ with ♠(src) = ♡(src) = l_1. After being processed at the cache, both packets arrive at the firewall. One of the pairs in the semantics of the firewall F is the following, with α unrestricted and $\beta(r) = 0$: $(\alpha \to (r{\leftarrow}0) \to \beta) \cdot \{$♡$[\mathsf{loadb/dst}]$, ♠$[\mathsf{loadb/dst}]\}$. After processing by the load balancer, both packets are sent to s_l simultaneously. To illustrate this event, we claim that there is a guarded pomset in the semantics of the load balancer. Observe that in the semantics of L we find the following pomset, with α and β from before (the second β is the result of the $r{=}0$ in L): $\alpha \to (r{\leftarrow}0) \to \beta \to \beta \to \{$♡$[s_l/\mathsf{dst}]$, ♠$[s_l/\mathsf{dst}]\}$. Using closure under contraction, we obtain a guarded pomset (the two β-nodes are merged into one) where both packets appear at s_l at the same time.

A final issue stems from the fact that the firewall implementation is flawed as written. Specifically, it uses a global variable to determine whether a packet should be forwarded on to a high priority host. Of course, if another packet arrives before the current one has been forwarded, the value of this variable might change, resulting in both packets being forwarded to a low priority host.

The issue with the firewall can be observed as follows. Take as input two packets ♠ and ♡ with ♠(src) = s_h and ♡(src) = s_l. After processing by the load balancer, both packets end up in the firewall. One of the pairs in the semantics of the firewall is the following, with $\alpha(v) = 1$ and β unrestricted: $(\alpha \cdot v{\leftarrow}0 \parallel \beta \cdot v{\leftarrow}1) \cdot \{$♡$[\mathsf{cache/dst}]$, ♠$[\mathsf{cache/dst}]\}$. After processing by the cache, both packets

Algorithm 1 Leader logic.

```
Initialize State:
  instance := 0
upon receiving pkt(msgtype, inst, rnd, vrnd, swid, value)
  match pkt.msgtype:
    case REQUEST:
      pkt.msgtype ← PHASE2A
      pkt.rnd ← 0
      pkt.inst ← instance
      instance := instance + 1
      multicast pkt to acceptors
    default :
      drop pkt
```

$$
\begin{array}{l}
\text{instance}{\leftarrow}0\,;\,\big(\\
\quad \mathit{msgtype} = \textsc{Request};\\
\quad \mathit{msgtype}{\leftarrow}\textsc{Phase2A};\\
\quad \mathit{rnd}{\leftarrow}0;\\
\quad \mathit{inst}{\leftarrow}\text{instance};\\
\quad \text{instance}{\leftarrow}\text{instance}+1;\\
\quad \mathit{dst}{\leftarrow}1 \| \cdots \| \mathit{dst}{\leftarrow}k\\
\big) + \mathsf{drop}
\end{array}
$$

Fig. 6: Leader logic from [9] and CNetKAT term, with k acceptors.

are sent to h_1 or l_1. To illustrate how the packets travel to e.g. l_1, we find the following pomset in the semantics of C, with α, β from before and $\gamma(v) = 0$:

$$\begin{tikzcd}
\alpha \arrow[r] & (v{\leftarrow}0) \arrow[dr] & & \\
& & \gamma \arrow[r] & \{\heartsuit[l_1/\mathsf{dst}], \spadesuit[l_1/\mathsf{dst}]\} \\
\beta \arrow[r] & (v{\leftarrow}1) \arrow[ur] & &
\end{tikzcd}$$

This pomset subsumes a guarded pomset. Hence, by exchange closure, we find guarded pomsets in the behavior of C where the packets both end up at l_1.

Overall, these examples show that CNetKAT can model subtle interactions between packets that arise in the presence of concurrency and state. Moreover, the axiomatic semantics can be used to prove (in)equivalences between programs.

6 Related Work

The core of CNetKAT is two extensions of Kleene Algebra: NetKAT [3,10], a networking extension of Kleene algebra with tests, and POCKA [37], a concurrent extension of KA. NetKAT describes how single packets move through a network, whereas CNetKAT can handle multiple packets. POCKA was introduced to describe concurrent interactions of global variables, whereas CNetKAT makes use of this algebra to enable intra-packet communication. CNetKAT captures local and global state interactions which was not in any of the previous work.

In the family of KA extensions, POCKA is closest to Concurrent Kleene algebra with Observations (CKAO) [15,16], which was proposed to integrate concurrency with conditionals such as if-statements and while-loops. Contrary to CKAO, which uses a Boolean algebra to axiomatize conditionals, POCKA uses a pseudocomplemented distributive lattice (PCDL) as the algebra for tests, which are referred to as observations to mark the difference. The idea to use a PCDL as the algebra for observations was first proposed in [14].

Our work fits within the CKA tradition, which gives a true concurrency semantics and is thereby distinct from bisimulation semantics typically considered in process algebras, such as CSP and CCS. Another distinction is that CNetKAT uses global state rather than message passing.

Some recently published work has also extended NetKAT with constructs for modeling multi-packet behavior [7]. Here the goal is to model interactions between the control- and date-plane in dynamic updates. Parallel composition is axiomatized with a left-merge operator and a communication-merge operator, and semantics is in terms of bisimilarity instead of traces. The examples largely focus on the table updates, not on the flow of packets through the network.

The current paper deviates from earlier concurrent variations on NetKAT, such as Concurrent NetCore [35] and a stateful variant of NetKAT introduced in [31]. Both have a different algebraic structure than NetKAT. Concurrent NetCore does not have Kleene star, and does not provide a denotational semantics, or axiomatization. Moreover, it does not handle multiple packets, the use of $+$ in the language is multicast rather than non-determinism, and $\|$ is concurrent processing of disjoint fields of the same packet. Because of these restrictions, concurrent NetCore is less suitable to specify inter-packet concurrency.

The approach in [31] models interactions among multiple packets, but is accompanied by semantic correctness guarantees, rather than algebraic formalizations as in CNetKAT. A recent PhD thesis [29] contains another version of stateful NetKAT, which assumes packet processing can always be serialized into a deterministic, global order. This assumption enables a simpler semantics and a decision procedure, though completeness is left as an open problem. Flow control in [29] is handled in the style of Guarded Kleene Algebra with Tests [22,36], which means that programs and specifications must be deterministic.

More broadly, there is a growing community doing research on network verification tools. Early work such as HSA [18], Anteater [30], Veriflow [19], Atomic Predicates [39], etc. focused on stateless SDN data planes, while more recent work such as p4v [27] and VMN [32] supports richer models such as P4 and stateful middleboxes. These tools typically use analyses based on symbolic simulation or they encode verification tasks into first-order formulas that can be checked using SMT solvers. To the best of our knowledge, CNetKAT is the first algebraic framework to model network-wide, multi-packet interaction with mutable state.

7 Discussion

We proposed CNetKAT, an algebraic framework to reason about programs with both local and global state, in the presence of parallel threads and control-flow statements. We provided a denotational semantics and a complete axiomatization. We also provided examples of how the language can be used to reason about stateful network programs and different sources of concurrency in a network.

As a result of the algebraic approach, the semantics of a program arises from the semantics of its parts. This clashes with the idea of observational equivalence when concurrency comes into play: some behaviors of a program can only be observed when executed concurrently with another program, and not in isolation. Hence it becomes necessary to include some elements in the semantics that do not immediately correspond to observable behavior. This implies that observational equivalence is not the right notion for axiomatising the semantics. However, using the greatest congruence contained in a notion of observational equivalence is interesting; this guided us in the development of our axiomatisation but it remains to be shown that our axiomatisation is indeed the greatest congruence.

CNetKAT relies on a classic approach to proving program correctness: develop a framework can model both specifications and implementations, and show that equivalence is decidable. Past experience with NetKAT suggests that this approach is usable, although CNetKAT lacks a procedure to check semantic equivalence, or at least membership of a given pomset. Devising an efficient procedure for this task is our immediate priority. The procedure will most likely rely on automata models such as fork automata [28] or Petri automata [6,5].

Ultimately, we would like to use CNetKAT to reason about stateful and distributed P4 programs. A target case study is provided in [9], which implemented Lamport's Paxos algorithm in the forwarding plane. To show correctness, the authors used a translation to Promela, a model checking language, and specify check that learners never decide on separate values for a single instance of consensus. This property is closely related to guarded pomsets. We would like to use CNetKAT to show correctness of the P4 implementation of the protocol directly (translation from the P4 code is almost direct, see Figure 6 for an example).

The reader will notice that the CNetKAT expression in Figure 6 uses an action of the form $f \leftarrow v$, where f is a field (*inst*) and v a global variable (instance). Adding actions of the converse form $v \leftarrow f$ is trivial since the packet logic specifies that f always has exactly one value. However, actions $f \leftarrow v$ require more care: the value of global variables can only be determined at the end since parallel threads might change it while it is being copied. To accommodate this in the semantics, we will have to allow partially defined packet fields and determine the missing field values at the end (when we check for guarded traces).

Another exciting direction for future work is the development of a library of *litmus tests* for networking in the spirit of [1]. Litmus tests are carefully crafted concurrent programs operating on shared memory locations that expose subtle bugs in memory models of hardware. One could imagine using the guarded pomsets semantics to discover minimal witnesses of undesired concurrent behavior.

We would also like to investigate the memory model of CNetKAT; this would give insight into the rules followed by operations on the global state. For a partial answer, we can look at POCKA. The guarded fragment of the POCKA semantics was shown to be *sequentially consistent* (concurrent memory accesses behave as if they are executed sequentially [24]), as it passed the *store buffering litmus test* [1]. The guarded fragment of the pomsets recording global variable changes is expected to pass this litmus test as well. It is worth investigating whether CNetKAT also supports other weak memory models, such as linearizability.

Acknowledgements N. Foster and T. Kappé were partially supported by DARPA grant HR001120C0107 (Pronto). T. Kappé also received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101027412 (VERLAN). D. Kozen was supported by NSF grant CCF-20008083. A. Silva was partially funded by ERC grant AutoProbe (101002697), EPSRC project CleVer (EP/S028641/1), and a Royal Society fellowship.

References

1. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Litmus: Running tests against hardware. In: TACAS. pp. 41–44 (2011). https://doi.org/10.1007/978-3-642-19835-9_5
2. Alpernas, K., Manevich, R., Panda, A., Sagiv, M., Shenker, S., Shoham, S., Velner, Y.: Abstract interpretation of stateful networks. In: Static Analysis. pp. 86–106. Springer International Publishing (2018), https://doi.org/10.1007/978-3-319-99725-4_8
3. Anderson, C.J., Foster, N., Guha, A., Jeannin, J., Kozen, D., Schlesinger, C., Walker, D.: NetKAT: semantic foundations for networks. In: POPL. pp. 113–126 (2014). https://doi.org/10.1145/2535838.2535862
4. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D.: P4: Programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev. **44**(3), 87–95 (jul 2014). https://doi.org/10.1145/2656877.2656890
5. Brunet, P., Pous, D.: Petri automata. Logical Methods in Computer Science **13** (02 2017). https://doi.org/10.23638/LMCS-13(3:33)2017
6. Brunet, P., Pous, D., Struth, G.: On decidability of concurrent Kleene algebra. In: CONCUR (2017), https://doi.org/10.4230/LIPIcs.CONCUR.2017.28
7. Caltais, G., Hojjat, H., Mousavi, M.R., Tunc, H.C.: DyNetKAT: An algebra of dynamic networks (2021), https://arxiv.org/abs/2102.10035
8. Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall, Ltd., London (1971)
9. Dang, H.T., Bressana, P., Wang, H., Lee, K.S., Zilberman, N., Weatherspoon, H., Canini, M., Pedone, F., Soulé, R.: P4xos: Consensus as a network service. IEEE/ACM Trans. Netw. **28**(4), 1726–1738 (2020). https://doi.org/10.1109/TNET.2020.2992106
10. Foster, N., Kozen, D., Milano, M., Silva, A., Thompson, L.: A coalgebraic decision procedure for netkat. In: POPL. pp. 343–355 (2015). https://doi.org/10.1145/2676726.2677011
11. Gischer, J.L.: The equational theory of pomsets. Theor. Comput. Sci. **61**, 199–224 (1988). https://doi.org/10.1016/0304-3975(88)90124-7
12. Grabowski, J.: On partial languages. Fundam. Inform. **4**(2), 427 (1981)
13. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra. In: CONCUR. pp. 399–414 (2009). https://doi.org/10.1007/978-3-642-04081-8_27
14. Jipsen, P., Moshier, M.A.: Concurrent Kleene algebra with tests and branching automata. J. Log. Algebr. Meth. Program. **85**(4), 637–652 (2016). https://doi.org/10.1016/j.jlamp.2015.12.005
15. Kappé, T., Brunet, P., Rot, J., Silva, A., Wagemaker, J., Zanasi, F.: Kleene algebra with observations. In: CONCUR. pp. 41:1–41:16 (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.41
16. Kappé, T., Brunet, P., Silva, A., Wagemaker, J., Zanasi, F.: Concurrent Kleene algebra with observations: From hypotheses to completeness. In: FOSSACS. pp. 381–400 (2020). https://doi.org/10.1007/978-3-030-45231-5_20
17. Kappé, T., Brunet, P., Silva, A., Zanasi, F.: Concurrent Kleene algebra: Free model and completeness. In: ESOP. pp. 856–882 (2018). https://doi.org/10.1007/978-3-319-89884-1_30
18. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: Static checking for networks. In: NSDI. pp. 113–126 (2012)

19. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: VeriFlow: Verifying network-wide invariants in real time. In: NSDI. pp. 15–29 (2013)
20. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inf. Comput. **110**(2), 366–390 (1994). https://doi.org/10.1006/inco.1994.1037
21. Kozen, D.: Kleene algebra with tests and commutativity conditions. In: TACAS. pp. 14–33 (1996). https://doi.org/10.1007/3-540-61042-1_35
22. Kozen, D., Tseng, W.D.: The Böhm-Jacopini theorem is false, propositionally. In: MPC. pp. 177–192 (2008). https://doi.org/10.1007/978-3-540-70594-9_11
23. Krob, D.: A complete system of B-rational identities. In: ICALP. pp. 60–73 (1990). https://doi.org/10.1007/BFb0032022
24. Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Computers **46**(7), 779–782 (1997). https://doi.org/10.1109/12.599898
25. Laurence, M.R., Struth, G.: Completeness theorems for bi-Kleene algebras and series-parallel rational pomset languages. In: RAMiCS. pp. 65–82 (2014). https://doi.org/10.1007/978-3-319-06251-8_5
26. Laurence, M.R., Struth, G.: Completeness theorems for pomset languages and concurrent Kleene algebras (2017), https://arxiv.org/abs/1705.05896
27. Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N., Foster, N.: p4v: Practical verification for programmable data planes. In: ACM SIGCOMM. pp. 490–503 (2018). https://doi.org/10.1145/3230543.3230582
28. Lodaya, K., Weil, P.: Series-parallel languages and the bounded-width property. Theoretical Computer Science **237**(1), 347–380 (2000). https://doi.org/10.1016/S0304-3975(00)00031-1
29. Long, X.: Primitives for Match-Action in Theory and Practice. Ph.D. thesis, Cornell University (2021)
30. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P.B., King, S.T.: Debugging the data plane with Anteater. In: SIGCOMM. pp. 290–301 (2011). https://doi.org/10.1145/2018436.2018470
31. McClurg, J., Hojjat, H., Foster, N., Cerný, P.: Event-driven network programming. In: PLDI. pp. 369–385 (2016). https://doi.org/10.1145/2908080.2908097
32. Panda, A., Lahav, O., Argyraki, K., Sagiv, M., Shenker, S.: Verifying reachability in networks with mutable datapaths. In: NSDI. pp. 699–718. USENIX Association, Boston, MA (Mar 2017)
33. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS (July 2002). https://doi.org/10.1109/LICS.2002.1029817
34. Salomaa, A.: Two complete axiom systems for the algebra of regular events. J. ACM **13**(1), 158–169 (1966). https://doi.org/10.1145/321312.321326
35. Schlesinger, C., Greenberg, M., Walker, D.: Concurrent netcore: From policies to pipelines. In: ICFP. p. 11–24 (Aug 2014). https://doi.org/10.1145/2628136.2628157
36. Smolka, S., Foster, N., Hsu, J., Kappé, T., Kozen, D., Silva, A.: Guarded Kleene algebra with tests: Verification of uninterpreted programs in nearly linear time. In: POPL (2020). https://doi.org/10.1145/3371129
37. Wagemaker, J., Brunet, P., Docherty, S., Kappé, T., Rot, J., Silva, A.: Partially observable concurrent Kleene algebra. In: CONCUR. pp. 20:1–20:22 (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.20

38. Wagemaker, J., Foster, N., Kappé, T., Kozen, D., Rot, J., Silva, A.: Concurrent NetKAT: modeling and analyzing stateful, concurrent networks (2022), https://arxiv.org/abs/2201.10485, full version of this article
39. Yang, H., Lam, S.S.: Real-time verification of network properties using atomic predicates. In: IEEE ICNP (2013), https://doi.org/10.1109/ICNP.2013.6733614