Concurrent NetKAT with Ports

Hui Feng
Leiden Institute of Advanced Computer Science
the Netherlands
h.feng@liacs.leidenuniv.nl

Marcello M. Bonsangue
Leiden Institute of Advanced Computer Science
the Netherlands
m.m.bonsangue@liacs.leidenuniv.nl

ABSTRACT

NetKAT is a powerful model extending Kleene algebra with tests (KAT) to programming networks. It supports the specification and reasoning about software-defined networks via automata-based operational semantics. This paper extends the NetKAT automata model to support concurrency using shared ports for communication and synchronization. We first extend the language of NetKAT protocols with communication actions and a parallel operator and give a closed and an open semantics using NetKAT automata. We show that NetKAT automata with an open semantics can be used as a model of the coordination language Reo via symbolic constraint automata.

CCS CONCEPTS

• **Theory of computation → Formal languages and automata theory**; **Concurrency**; • **Networks**;

KEYWORDS

Concurrent NetKAT, Reo, Constraint automata, Software-defined networks

ACM Reference Format:
Hui Feng and Marcello M. Bonsangue. 2024. Concurrent NetKAT with Ports. In *Proceedings of ACM SAC Conference (SAC'24).* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3605098.3636048

1 INTRODUCTION

The rapid evolution of technology, increasing network traffic, and the need for flexible and scalable computer networks have necessitated a paradigm shift in network management. Traditional network architectures use distributed switches to receive and forward packets, each switch consisting of hardware and dedicated control software. Software Defined Networks (SDNs) provide a centralized approach to network control and management by separating the control plane from the data plane [14]. This separation allows for programmability and agility in network configurations, enabling dynamic provisioning of resources, efficient traffic management, and the ability to adapt to changing requirements.

The level of programmability of the software controllers in an SDN to handle traffic flow, routing decisions, and network policies together with the use of protocols such as OpenFlow [22] have generated increasing interest in the academic community to provide a theoretical foundation for understanding the principles, components, and interactions within SDNs. Examples include model-checking to verify controller programs [1, 5, 6, 12], formal models of OpenFlow [11, 18], or some specific part of it, such as the topology discovery mechanism [26] or security protocols [9].

Different from other process algebras like CSP [15], a policy-based approach is taken by NetKAT [2], a model that emphasizes the policy-driven nature of SDNs. It consists of an extension with variables of Kleene Algebra with Tests tailored to define high-level policy specification and network components and observe the network behavior from the point of view of a packet [19]. NetKAT, however, is not stateful and does not allow modeling concurrent policies and multiple packets. In this paper, we present pNetKAT, a conservative extension of NetKAT, allowing multiple concurrent policies to communicate via shared ports. In pNetKAT, ports are treated as shared variables that can be undefined when no communication is possible. We give an operational semantics to pNetKAT using non-deterministic NetKAT automata with a slightly modified acceptance rule that enforces observability only of sequences with successful synchronization steps). Without ports, both syntactically and semantically pNetKAT and NetKAT coincide.

Under the assumption that ports are declared as either input or output, we give another semantics to pNetKAT by refining the acceptance rule of non-deterministic NetKAT automata to allow for the system to interact with the environment along the input and output ports. The new semantics is an extension of the previous one (and thus the new equivalence is stricter, in general). We show that this model can be used as semantics for the coordination language Reo [3]. from which we can borrow the join composition operator and define it for NetKAT automata with input and output ports.

Unlike other methods, our pNetKAT extension to a stateful and concurrent NetKAT is conservative as it remains in the semantic realm of language equivalence instead of moving to pomset [24] or bisimulation equivalence [8]. The connection with Reo paves the way to a more expressive concurrent NetKAT, with (concurrent, stateful) policies declaring input and output ports (as switches and controllers in SDNs) that can be composed using a join operation (only communication on common ports must synchronize, while policies using undeclared ports in another process can proceed in parallel).

We proceed as follows. In Section 2 we briefly present NetKAT with a focus on the automata model. While the original model is deterministic we present also an equivalent but more compact model based on non-deterministic NetKAT automata (NKA). In Section 3 we extend NetKAT protocols with communication actions and concurrency and define a closed semantics using non-deterministic NetKAT automata with ports (*pNKA*). We continue in Section 4 by introducing non-deterministic NetKAT automata with input and output ports (*ioNKA*) and use them to model NetKAT with ports. We then briefly recall Reo and its symbolic constraint automata semantics and show how to compositionally translate them into NetKAT automata.

Related work

There are several works extending NetKAT in different directions. For example, [21] introduces network event structures to model constraints on updates and define an extension of NetKAT policies with mutable state to give semantics to stateful SDN controllers. DyNetKAT [8] is a NetKAT extension with concurrency and a stateful state to model SDNs with dynamic configurations. The extended language is a process algebra with constructs for synchronization, sequential composition, and recursion built on top of NetKAT policies. While DyNetKAT allows for multi-packet behavior, the syntax does not allow for the basic NetKAT "dup" action. Also, the focus is on bisimulation rather than our (and NetKAT) language equivalence, which comes equipped with sound and ground-complete axiomatization.

Staying in the realm of Kleene algebra is the line of works followed by [24], where CNetKAT is introduced as a combination of Kleene algebra with tests, concurrent Kleene algebra, and network operators. The semantics is given in terms of pomset languages and is thus based on true concurrency rather than interleaving.

Besides the work we already mentioned, there are other formal models for SDN closely related to NetKAT that involve concurrency. For example, concurrent NetCore [23] extends NetCore with concurrency, while NetKAT is an extension of NetCore with Kleene star. In terms of tools, SDNRacer [10] checks various concurrency-induced errors in SDNs and precisely captures the asynchronous interaction between controllers and switches.

Constraint automata are the first automata-based model for Reo connectors [4]. Since then, various other operational models have emerged (see [17] for an overview). Relevant to our work here is the extension of constraint automata with memory [16] and the more recent work of symbolic constraint automata [12] that focus on an implementable subset, instead of an efficient computation of the composition operator. In this paper, we show how to embed symbolic constraint automata into *ioNKA*. We follow I/O automata [20] and constraint automata [4] by explicitly declaring at the interface the ports that are used as input and output. Transitions in *ioNKA*, however, are neither action-based nor imperative, but rather declarative using pre- and post-conditions in the style of NetKAT automata.

2 NETKAT

In this section, we briefly introduce NetKAT [2], a language for specifying the flow of a packet through a network, and give its semantics in terms of finite automata and languages.

We assume fixed a finite set of fields Fld, say of size k, and a finite set of values Val. A packet π is a record of fields, that is, a function from Fld to Val that we represent by $[f_1 = v_1, \cdots, f_k = v_k]$. Tests for the value stored in a field form the basic building block for the set of predicates $B(Fld)$ defined by the following grammar:

$$a, b ::= 1 \mid 0 \mid f = v \mid a + b \mid a \cdot b \mid \neg a\,.$$

The set of all predicates (modulo the usual equations) forms a Boolean algebra, where $+$ is interpreted as the disjunction, $\cdot$ as the conjunction, and $\neg$ as negation. Further, 1 is the truth predicate, and 0 denotes false. The set At of atoms α, β of the Boolean algebra $B(Fld)$ corresponds to the set of valuations, that is complete conjunctions of basic tests $f = v$ ranging over all fields in Fld. For simplicity, and with a convenient abuse of notation, we denote an atom as a record $\alpha = [f_1 = v_1, \cdots, f_k = v_k]$, allowing us to switch between packets and atoms. The behavior of a packet through the network is specified by policies

$$p, q ::= a \mid f \leftarrow v \mid dup \mid p + q \mid p \cdot q \mid p^*\,.$$

Here a is a predicate in $B(Fld)$, $f \leftarrow v$ is the assignment of the value v to the field f of a packet, $p + q$ is the nondeterministic choice between the policies p and q, $p \cdot q$ specify the sequential composition of two policies, and p^* the iterative execution of a policy p. The predicate 0 denotes failure and 1 is skip. As usual, we will often not write "·" in policies. When applied to predicates, "+" and "·" act as disjunctions and conjunctions, respectively.

The behavior of a packet π through the network is specified by a string in $(At \cdot At) \cdot At^*$, denoting a sequence of conditions satisfied by the packet π before and after being forwarded from one switch to another in the network. Syntactically, the forwarding is specified by the action dup, which is thus the only observable action of a policy. The semantics of a policy is then given by the set of all possible behaviors of a packet under that policy. Since this is a regular subset of $(At \cdot At) \cdot At^*$, following [13], we use an automaton to describe it.

Definition 2.1. A deterministic NetKAT automaton (dNKA) is a tuple $(S, Fld, \delta, \xi, s_0)$ where

- S is a finite set of states,
- Fld is a finite set of fields,
- $\delta\colon S \times At \times At \to S$ is a transition map,
- $\xi\colon S \times At \times At \to 2$ is an observation map, and
- $s_0 \in S$ is a distinguished initial state.

Here At is the set of atoms of $B(Fld)$, and 2 is the two-element Boolean set.

Differently from an ordinary automaton, a dNKA uses pre- and post-conditions as labels to specify the execution of an action in a computation. Here $\delta(s, \alpha, \beta) = s'$ denotes a transition from state s to a state s' executed by an action satisfying the pre-condition α and resulting in a post-condition β. Further, the observation map $\xi(s, \alpha, \beta) = 1$ if and only if an action in state s satisfies the pre-condition α, results in the post-condition β, and successfully terminates a computation.

Figure 1 shows a dNKA. There are four states but only $\{s_0, s_1, s_2\}$ are accepting computations that end in the pair of atoms labeling the respective vertical down arrows. The state s_0 is the initial state, as marked by an incoming arrow without a source. As usual, labeled arrows between two states represent the transition map. Here we assume only three atoms: α, β, and γ.

The language accepted by a dNKA is a subset of strings in $(At \cdot At) \cdot At^*$ and is defined with the help of the following auxiliary acceptance predicate:

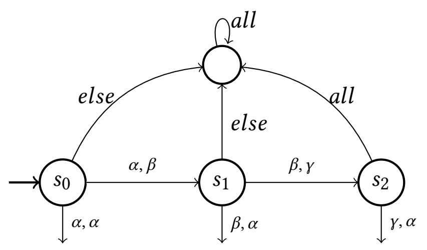

Figure 1: An example of a dNKA

Definition 2.2. For a dNKA $M = (S,\ Fld,\ \delta,\ \xi,\ s_0)$, we say that a string $\sigma \in (At \cdot At) \cdot At^*$ is accepted by M if and only if the deterministic acceptance predicate $DAcc(s_0, \sigma)$ holds, where $DAcc$ is defined inductively as follows:

- $DAcc(s, \alpha\beta) = \xi(s, \alpha, \beta)$,
- $DAcc(s, \alpha\beta \cdot \tau) = DAcc(\delta(s, \alpha, \beta), \beta \cdot \tau)$,

where $s \in S$, $\alpha, \beta \in At$, and $\tau \in At^+$. The language $L_d(M)$ is defined as the set of all strings accepted by M.

The language of the automaton in Figure 1 is $\{\alpha\alpha, \alpha\beta\alpha, \alpha\beta\gamma\alpha\}$. In fact, for example, $DAcc(s_0, \alpha\beta\alpha) = DAcc(s_1, \beta\alpha) = \xi(s_1, \beta, \alpha) = 1$.

For a more compact representation of the operational semantics of NetKAT, we use non-deterministic NetKAT automata as introduced in [25].

Definition 2.3. A non-deterministic NetKAT automaton (NKA) is a tuple $(S,\ Fld,\ \Delta,\ \Xi,\ s_0)$, where

- S is a finite set of states;
- Fld is a finite set of fields;
- $\Delta \colon S \times At \times At \to \mathcal{P}(S)$ is a transition relation;
- $\Xi \colon S \times At \times At \to 2$ is an observation map, and
- $s_0 \in S$ is a distinguished initial state.

As before, here At is the set of atoms of $B(Fld)$.

For example, the sub-automaton defined by restricting the one in Figure 1 to the three states s_0, s_1 and s_2 is an NKA.

Having non-determinism is reflected in the definition of the language accepted, which now selects only transitions leading to successful computations.

Definition 2.4. For an NKA $N = (S,\ Fld,\ \Delta,\ \Xi,\ s_0)$, we say that a string $\sigma \in (At \cdot At) \cdot At^*$ is accepted by N if and only if the non-deterministic acceptance predicate $NDAcc(s_0, \sigma)$ holds, where $NDAcc$ is defined inductively as follows:

- $NDAcc(s, \alpha\beta) = \Xi(s, \alpha, \beta)$,
- $NDAcc(s, \alpha\beta \cdot \tau) \iff \exists s' \in \Delta(s, \alpha, \beta)\,.\,NDAcc(s', \beta \cdot \tau)$,

where $s \in S$, $\alpha, \beta \in At$, and $\tau \in At^+$. The language $L_{nd}(N)$ is defined as the set of all strings in $(At \cdot At) \cdot At^*$ accepted by N.

Every dNKA can be easily seen as an NKA with a functional transition relation. Conversely, given an NKA, we can construct a dNKA that is language equivalent.

Theorem 2.5. *For every NKA N there exists a dNKA M such that $L_d(M) = L_{nd}(N)$.*

The result is similar to the powerset construction for ordinary finite automata. In fact, given a NKA $N = (S,\ Fld,\ \Delta,\ \Xi,\ s_0)$ we can define a dNKA $M = (\mathcal{P}(S),\ Fld,\ \delta,\ \xi,\ \{s_0\})$ with

- $\xi(X, \alpha, \beta) = 1$ if and only if $\exists s \in X.\Xi(s, \alpha, \beta) = 1$,
- $s \in \delta(X, \alpha, \beta)$ if and only if $\exists s' \in X.s \in \Delta(s', \alpha, \beta)$.

Then, for all $X \subseteq S$, $\alpha, \beta \in At$, and $\sigma \in At^*$ we can prove that $DAcc(X, \alpha\beta{\cdot}\sigma)$ if and only if there exists $s \in X$ such that $NDAcc(s, \alpha\beta{\cdot}\sigma)$. Note that the above language equivalence does not hold if Δ and Ξ would take as input general Boolean predicates instead of atoms.

In Table 1 we give the operational semantics of NetKAT policies in terms of an NKA. States of the automaton are policies themselves, that we consider modulo associativity, idempotency, and commutativity of the "+" operation to guarantee local finiteness. Basically, a state represents (an equivalence class of) what still needs to be executed.

Table 1: Operational semantics of NetKAT

$dup \xrightarrow{\alpha,\alpha} \alpha$	$\dfrac{\alpha \le a}{a\downarrow_{(\alpha,\alpha)}}$	$\dfrac{\beta \le (f = v)}{f \leftarrow v\downarrow_{(\alpha,\beta)}}$
$\dfrac{p_1 \xrightarrow{\alpha,\beta} p}{p_1 + p_2 \xrightarrow{\alpha,\beta} p}$		$\dfrac{p_1\downarrow_{(\alpha,\beta)}}{p_1 + p_2\downarrow_{(\alpha,\beta)}}$
$\dfrac{p_1 \xrightarrow{\alpha,\beta} p}{p_1 \cdot p_2 \xrightarrow{\alpha,\beta} p \cdot p_2}$		$\dfrac{p_1\downarrow_{(\alpha,\beta)} \quad p_2\downarrow_{(\beta,\gamma)}}{p_1 \cdot p_2\downarrow_{(\alpha,\gamma)}}$
$\dfrac{p_1\downarrow_{(\alpha,\beta)} \quad p_2 \xrightarrow{\beta,\gamma} p}{p_1 \cdot p_2 \xrightarrow{\alpha,\gamma} p}$		
$\dfrac{p \xrightarrow{\alpha,\beta} p_1}{p^* \xrightarrow{\alpha,\beta} p_1 \cdot p^*}$		$p^*\downarrow_{(\alpha,\alpha)}$
$\dfrac{p\downarrow_{(\alpha,\beta)} \quad p^* \xrightarrow{\beta,\gamma} p_1}{p^* \xrightarrow{\alpha,\gamma} p_1}$		$\dfrac{p\downarrow_{(\alpha,\beta)} \quad p^*\downarrow_{(\beta,\gamma)}}{p^*\downarrow_{(\alpha,\gamma)}}$

We have two types of rules: those specifying transitions (on the left-hand side of Table 1), and those for observations, specifying the accepting states (on the right-hand side). Intuitively, the behavior of a policy is to guide a given packet into a network. This is described by the assignment of values to the fields to record, for example, where the packet is, where it has to go, and other information. Policies filter out executions via predicates. The basic transition step of a policy is given only by the execution of a *dup* action. Predicate evaluations and field assignments are evaluated locally in the current state. A policy execution may terminate in an accepting state (as specified on the right-hand side of Table 1) or may diverge in an infinite computation (via the transition rules of p^*) and not be observed. Note that since we consider states modulo associativity, commutativity, and idempotency of the "+" operation, there is no need for symmetric rules for the "+" for both the transition and the observation relation.

For a given policy p, in [13] a dNKA $M(p)$ is constructed using syntactic derivatives. Similarly, Let $N(p)$ denote the NKA constructed using the rules in Table 1, with as initial state (the equivalence class of) p. We then have the automata $M(p)$ and $N(p)$ accept the same language [25].

3 NETKAT WITH PORTS

Next, we extend NetKAT protocols with a parallel operator and allow policies to communicate via ports. A port x is a shared variable between two processes that can be updated with a value v by an output operation $x!v$ and can be destructively read by an input operation $x?f$ which stores the communicated value into a field f. Unlike a variable, however, a port may be undefined, here denoted by the symbol $\perp$ that we assume is not a value in *Val*. Intuitively, a port x is undefined, i.e. $x = \perp$, if it can be used by an output operation. Dually, input on a port x can only take place if x is not undefined, i.e. $\neg(x = \perp)$ that, as usual, we denote by $x \neq \perp$. In other words, we see an output $x!v$ as the atomic execution of the guarded command $x = \perp \cdot x \leftarrow v$, whereas an input $x?f$ can be seen as the atomic execution of the guarded command $x \neq \perp \cdot f \leftarrow x \cdot x \leftarrow \perp$. Here we use the assignment $f \leftarrow x$ of a variable to a field, which is just an abbreviation for the protocol $\Sigma_{v\in Val}(x = v \cdot f \leftarrow v)$ because Val is assumed to be finite. Communication of two parallel protocols via a port x in an undefined state is then the atomic execution of an output command on x followed by an input on x, resulting in the command

$$(x = \perp \cdot x \leftarrow v) \cdot (x \neq \perp \cdot f \leftarrow v \cdot x \leftarrow \perp)$$

which, because is executed atomically, can be thought of as equivalent to $x = \perp \cdot f \leftarrow v \cdot x \leftarrow \perp$.

Formally, we assume a finite set of variables *Var* partitioned in a set of fields *Fld* and a set of ports *Prt*. As for NetKAT, fields are ranged over by f, while ports are ranged by x. All variables can store values from *Val* but only ports can be undefined, which we denote with $\perp \notin Val$. The set of predicates $B(Var)$ extends those of NetKAT by allowing basic tests on all variables, including ports, as defined by the grammar

$$a, b ::= 1 \mid 0 \mid f = v \mid x = v \mid x = \perp \mid a + b \mid a \cdot b \mid \neg a,$$

where, $f \in Fld$, $x \in Prt$, and $v \in Val$. We use $f = x$ as a shorthand for the test $\Sigma_{v\in Val} x = v \cdot f = v$. This is well defined because the set *Val* is finite. The behavior of a packet in pNetKAT through a network subject to several communicating parallel policies is specified by the following grammar that extends the one of NetKAT with communication actions and a parallel operator:

$$p, q ::= a \mid f \leftarrow v \mid dup \mid x?f \mid x!v \mid p + q \mid p \cdot q \mid p||q \mid p^* .$$

As discussed above, here $x?f$ is an input action that is executed only when the port x has a value available that is assigned immediately to the field f. The output action $x!v$ is executed if the port x is not busy (there is no value) and makes available the value v at the port. Note that only fields can be assigned directly by policies, whereas ports can change values only through successful communications. Policies can be executed in parallel via the operator "||". Parallel policies executing an input, respectively an output, action on the same port synchronize.

The operational semantics of pNetKAT is given in terms of NKA as presented in Definition 2.3. The only addition to the rules given in Table 1 is the transition and observation map for input and output actions and for the parallel composition of policies. The extra rules are presented next.

Input and output actions are, like dup, primitive actions that have a transition step and do not terminate for any observable pairs of atoms:

$$\frac{\alpha \leq (x = v) \quad \beta = \alpha[\perp/x][v/f]}{x?f \xrightarrow{\alpha,\beta} \beta} \qquad \frac{\alpha \leq (x = \perp) \quad \beta = \alpha[v/x]}{x!v \xrightarrow{\alpha,\beta} \beta}$$

The conditions in the premises of the two rules express the precondition and postcondition of the input and output, respectively, as we already discussed. Here $\alpha[v/x]$ ($\alpha[v/f]$) is the atom assigning a port x to v (a field f to v, respectively) and all other variables are as in α.

The transition relation of the parallel composition $p_1||p_2$ of two policies p_1 and p_2 is described by three types of rules, namely: synchronization, interleaving, and termination. When they occur in parallel, an input and an output action on the same ports synchronize:

$$\frac{p_1 \xrightarrow{\alpha_1,\beta_1} p \quad p_2 \xrightarrow{\alpha_2,\beta_2} q}{p_1||p_2 \xrightarrow{\alpha,\beta} p||q} \qquad \frac{p_1 \xrightarrow{\alpha_1,\beta_1} p \quad p_2 \xrightarrow{\alpha_2,\beta_2} q}{p_2||p_1 \xrightarrow{\alpha,\beta} q||p}$$

under the condition that there is a port $x \in Prt$ and a field $f \in Fld$ such that $\alpha(x) = \beta(x) = \alpha_1(x) = \beta_2(x) = \perp$ and $\beta_1(x) = \alpha_2(x) = \beta_2(f)$, whereas for all other variables $y \in Var$ different from x, $\alpha_1(y) = \alpha_2(y) = \alpha(y)$ and $\beta_1(y) = \beta_2(y) = \beta(y)$. The above condition says that the pair (α_1, β_1) describes the output of the value v on a port x, that is received and assigned to field f by the input action specified by (α_2, β_2). For all other variables, the preconditions and the postconditions of all transitions involved do not change.

If the transition of a policy does not have a visible effect on the state of a port, then when in parallel with any other policy it can proceed in an interleaving fashion:

$$\frac{p_1 \xrightarrow{\alpha,\beta} p}{p_1||p_2 \xrightarrow{\alpha,\beta} p||p_2} \qquad \frac{p_1 \xrightarrow{\alpha,\beta} p}{p_2||p_1 \xrightarrow{\alpha,\beta} p_2||p}$$

where $\alpha(x) = \beta(x)$ for all port $x \in Prt$. Note that, the above symmetric rules in combination with the synchronization rules imply that there cannot be multiparty synchronization.

Similar to the shuffle of languages, if a policy p_1 terminates when in parallel with another policy p_2, then p_2 can continue alone from the postcondition observed at the termination of p_1:

$$\frac{p_1{\downarrow}(\alpha,\beta) \quad p_2 \xrightarrow{\beta,\gamma} p}{p_1||p_2 \xrightarrow{\alpha,\gamma} p} \qquad \frac{p_1{\downarrow}(\alpha,\beta) \quad p_2 \xrightarrow{\beta,\gamma} p}{p_2||p_1 \xrightarrow{\alpha,\gamma} p} \qquad \frac{p_1{\downarrow}(\alpha,\beta) \quad p_2{\downarrow}(\alpha,\beta)}{p_1||p_2{\downarrow}(\alpha,\beta)}$$

Generally, the parallel composition of two policies does not terminate immediately, as it may involve input and output actions. However, if no communication action is involved, then it terminates observing the pair (α, β) if both policies do the same. Note that this means inconsistent policies cannot terminate successfully, as they both act atomically on the same packet.

As in the previous section, we denote by $N(p)$ the NKA constructed using the rules in Table 1 and the above ones for the parallel composition, with as states equivalence classes of policies modulo commutativity and associativity of both "+" and "||", and idempotency of only "+", and with as initial state (the equivalence class of) p. To enforce synchronization, we impose that ports are undefined at all times in every accepted string (a condition satisfied by the synchronization step but not by the postcondition of an open output and a precondition of an open input). We thus refine the acceptance predicate for NKA with ports (thus, pNetKAT) as follows:

Definition 3.1. Let At be the set of atoms of the Boolean algebra $B(Var)$. For an NKA (with ports) $N = (S,\ Var,\ \Delta,\ \Xi,\ s_0)$, we say that a string $\sigma \in (At \cdot At) \cdot At^*$ is accepted by N if and only if the predicate $PAcc(s_0, \sigma)$ holds, where $PAcc$ is defined inductively as follows:

- $PAcc(s, \alpha\beta) \iff \Xi(s, \alpha, \beta)$ and $\forall x \in Prt.\alpha(x) = \beta(x) = \bot$,
- $PAcc(s, \alpha\beta\sigma) \iff \exists s' \in \Delta(s, \alpha, \beta)\,.\, PAcc(s', \beta\sigma)$ and $\forall x \in Prt.\alpha(x) = \bot$,

where $s \in S$, $\alpha, \beta \in At$, and $\sigma \in At^+$. The language $L_p(N)$ is defined as the set of all strings in $(At \cdot At) \cdot At^*$ accepted by N. We refer to NKA with $PAcc$ predicates as *pNKA*.

Because of the symmetry in the rules of the parallel composition, we have that "||" is a commutative and associative operator. It is not idempotent in general, except for policies with no occurrences of dup, input, or output actions. For example $a||b = a \cdot b$ and $f \leftarrow v||f \leftarrow v = f \leftarrow v$.

Clearly, if there are no ports in Var (i.e. $Var = Fld$) then they do not appear in atoms in At. In this case, the definition of $PAcc$ coincides with the usual definition $NDAcc$ of accepted strings for an NKA. Note that because ports are undefined in every atom occurring in a string accepted by $PAcc$ ports can be removed (or added) to an NKA without changing its language equivalence. Using the Kleene theorem for NetKAT [13], we can relate (non-compositionally) pNetKAT with NetKAT:

Theorem 3.2. *For every pNetKAT policy p there is a NetKAT policy q such $L_{nd}(N(q))$ is equal to $L_p(N(p))$ after removing the ports from every atom.*

This implies that for every process in pNetKAT we can find an 'equivalent' process in NetKAT, basically by compiling parallel processes into interleaved ones if no open communication is involved and transforming synchronizations into assignments. In other words, the semantics of pNetKAT is a closed semantics not allowing any external communication after the system is defined. In the next section, we define an open semantics that allows for the synchronization of several ports at the same time.

We conclude this section with an example adapted from [8] and sketched in Figure 2. Two switches SX and SY have 3 ports each: $x1, x2, x3$ and $y1, y2, y3$, respectively. Their behavior depends on their current flow table and it is described by the following set of policies:

$$\begin{array}{ll} SX_0 = 0 & SY_0 = 0 \\ SX_1 = (f = x1) \cdot f \leftarrow x3 & SY_1 = (f = y3) \cdot f \leftarrow y1 \\ SX_2 = (f = x2) \cdot f \leftarrow x3 & SY_2 = (f = y3) \cdot f \leftarrow y2, \end{array}$$

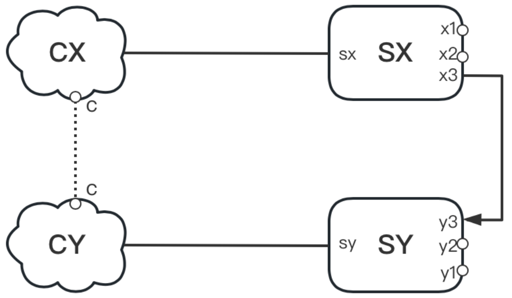

Figure 2: A SDN with two switches and two controllers

where f is a field of a packet that records the last passed port. The switches are linked through ports $x3$ and $y3$:

$$L = (f = x3) \cdot dup \cdot f \leftarrow y3\,.$$

Under the flow tables SX_1 and SY_1, for example, a packet that arrives at port $x1$ of switch SX is forwarded to port $x3$. The latter is linked to port $y3$ of switch SY, which forwards the packet to port $y1$. Note the role of the dup action to record that a packet moves from one switch to another.

Each switch is linked with a controller via the ports sx and sy. CX is the controller of switch SX and CY of switch SY. The two controllers are concurrently acting on their switch by updating their flow tables. The task of the two controllers is to guarantee that incoming packets at port $x1$ arrive at port $y1$ and incoming packets at port $x2$ arrive at port $y2$. No mixing of flow is allowed. To avoid race conditions, the controllers have to synchronize and guarantee a proper order of execution of their concurrent behaviors:

$$\begin{array}{rcl} CX & = & (f = x1) \cdot (sx!0 \cdot c!1 \cdot c?g \cdot sx!1) \\ & & + (f = x2) \cdot (sx!0 \cdot c!2 \cdot c?g \cdot sx!2) \\ CY & = & c?g \cdot ((g = 1 \cdot sy!1) + (g = 2 \cdot sy!2)) \cdot c!0\,. \end{array}$$

Here sx and sy are the ports connecting the controllers to their controlled switches. When sending the flow message 0, 1, or 2, the flow table will be updated accordingly. The two controllers use port c to synchronize each other and pass the information about which flow table they have updated. While waiting for the update of the flow table of switch SY, the switch SX first drops all incoming packets, and only after SY is updated then SX accept packets from the correct port.

The behavior of the entire network is given by

$$(ft1 \leftarrow 0) \cdot (ft2 \leftarrow 0) \cdot (N^*||CX||CY)^*,$$

where

$$N = \Sigma_{j=0}^{2}(ft1 = j) \cdot SX_j + \Sigma_{j=0}^{2}(ft2 = j) \cdot SY_j + L + sx?t1 + sy?t2$$

Initially, both switches start with empty flow tables that are updated when a controller sends a flow message to its switch via the port sx or sy, respectively.

4 NETKAT AUTOMATA WITH I/O PORTS

In the previous section, we used NKA for giving a closed semantics of our concurrent policy language pNetKAT using the acceptance predicate $PAcc$ that takes into account ports. Next, we consider NetKAT automata for open concurrent systems and use them as a model of pNetKAT.

To begin with, we partition the set of ports *Prt* into input ports *IPrt* and output ports *OPrt*. Together with the disjoint set of fields *Fld* they form a finite set of variables *Var*. Input ports are ranged over by i and output ports by o. As before, all variables can store values from *Val* but only input and output ports can be undefined, which we denote with $\bot \notin Val$. Intuitively, an input port i of a connector is enabled if it contains a value different from $\bot$ so that this value is ready to be taken by the connector when synchronizing on i with the environment that puts the value in it. Dually, an output port o of a connector is undefined (i.e., $o = \bot$) when the port o is ready to receive a value from the connector and synchronizes with the environment when it will read from o.

We use input and output ports to define a novel operational behavior of NKA by an acceptance predicate that, differently from *PAacc*, does not enforce synchronization and leaves the system open to communication instead of closing it in the style of [7].

Definition 4.1. Let At be the set of atoms of the Boolean predicates $B(Var)$, where $Var = IPrt \cup OPrt \cup Fld$. For an NKA $N = (S,\ Val,\ \Delta,\ \xi,\ s_0)$ with atoms At involving input and output ports, we say that a string $\sigma \in (At \times At)^+$ is accepted by N if and only if the predicate $IOAcc(s_0, \sigma)$ holds, where *IOAcc* is defined inductively as follows:

- $IOAcc(s, (\alpha, \beta)) \iff \Xi(s, \alpha, \beta)$,
- $IOAcc(s, (\alpha, \beta)\sigma) \iff \exists s' \in \Delta(s, \alpha, \beta) \,.\, IOAcc(s', \sigma)$ and $\beta \triangleright head(\sigma)$,

where $s \in Q$, $\alpha, \beta \in At$, $\sigma \in (At \times At)^+$. The language $L_{IO}(N)$ is defined as the set of all strings in $(At \times At)^+$ accepted by N. We refer to NKA with *IOAcc* predicates as *ioNKA*

A pair (α, β) in a string accepted above represents the pre/post condition of an action executed by a component. In between two pairs, the environment can communicate with the components and change the values at its ports. We formalize this using the $\triangleright$ predicates. In fact, for every string in $(At \times At)^+$, we define $head((\alpha, \beta)\sigma) = \alpha$, and for every two atoms α and β we say that the predicate $\beta \triangleright \alpha$ holds if and only if:

a. local variables cannot be modified by the environment, i.e., $\beta(f) = \alpha(f)$ for every field $f \in Fld$;
b. the environment can put a value to an input port only if the port is not already enabled, i.e. either $\beta(i) = \alpha(i)$ or $\beta(i) = \bot$;
c. the environment can take a value from an output port only if there is one, i.e., either $\beta(o) = \alpha(o)$ or $\alpha(o) = \bot$.

Here we see β as the postcondition of an action, and α as the precondition of the next action both to be executed by the component, or, dually, they are the pre- and postcondition of actions executed by the environment. The conditions on the second and third items above allow the environment to communicate with a component only through input ports that are not enabled and output ports that contain values. As such the semantics of a component caters to all possible interactions with the environment and is open. For example, if a component executes an action ending in a postcondition $[f = 1, i = \bot, o = 3]$ then the environment could assign a value to the input port i so that at the next step the component would start with a precondition $[f = 1, i = 2, o = 3]$. Alternatively, the environment could take the value from the output port o and put a value in the input variable i resulting in the next step component precondition $[f = 1, i = 2, o = \bot]$. However, the environment could never change the value of the field f as it is local to the component.

The set of input and output ports used by a pair (α, β) is defined by

$$I(\alpha, \beta) = \{i \in IPrt \mid \alpha(i) \neq \beta(i) = \bot\} \text{ and}$$
$$O(\alpha, \beta) = \{o \in OPrt \mid \beta(o) \neq \alpha(o) = \bot\}\,.$$

The above reflects the fact that an input port must be enabled in the precondition and is available for communication after the value has been taken, and dually for an output port.

In the absence of input and output ports, the condition on the first item ensures that for any two consecutive pairs $(\alpha_1, \beta_1)(\alpha_2, \beta_2)$ occurring in an accepted string, the postcondition β_1 is equal to the precondition α_2. In this case, we can transform a strings $\sigma \in (At \times At)^+$ into essentially equal strings in $t(\sigma) \in (At \cdot At) \cdot At^*$ as follows:

$$t((\alpha, \beta)) = \alpha\beta \quad t((\alpha, \beta)\sigma) = \alpha \cdot t(\sigma)\,.$$

The transformation t unifies the subsequent postcondition and precondition because they are equal. The inverse t^{-1} of t maps strings in $(At \cdot At) \cdot At^*$ into strings in $(At \times At)^+$ by equating subsequent postcondition and precondition:

$$t^{-1}(\alpha\beta) = (\alpha, \beta) \quad t^{-1}(\alpha, \beta\sigma) = (\alpha, \beta) \cdot t^{-1}(\beta\sigma)\,.$$

Here $\sigma \in At^+$ and α, β are atoms in $B(Var)$, with $Var = IPrt \cup OPrt \cup Fld$.

Theorem 4.2. *For every NKA automaton with no (input and output) ports, $IOAcc(s, \sigma) = PAcc(s, t(\sigma)) = NDAcc(s, t(\sigma))$, for any state s and string $\sigma \in (At \times At)^+$.*

In other words, the predicate *IOAcc* is a conservative extension of *NDAcc* in the context of NetKAT automata when there are no ports. However, if we assume $Prt = IPrt \cup Oprt$ and $Var = Prt \cup Fld$ so that atoms in $B(Var)$ are of the correct type for both predicates *PAcc* and *IOAcc*, we then have the following result.

Theorem 4.3. *Let $Var = Prt \cup Fld$ and $Prt = IPrt \cup Oprt$ and $(S,\ Var,\ \Delta,\ \Xi,\ s_0)$ be a NKA. For every string $\sigma \in (At \cdot At) \cdot At^*$ where At is the set of atoms of $B(Var)$ and $s \in S$ if the predicate $PAcc(s, \sigma)$ holds then also $IOAcc(s, t^{-1}(\sigma))$ holds.*

As a consequence of the above, we have that if two policies of pNetKAT are language equivalent with respect to the *IOAcc* then they are also language equivalent with respect to *PAcc*. The converse is in general not true, meaning that the equivalence generated by *pNKA* is coarser than that of *ioNKA*.

4.1 Reo and symbolic constraint automata

Next, we show that NetKAT automata can be used to express the semantics of the coordination language Reo [3] too. Reo is a formalism that allows for the specification and composition of complex concurrent systems by focusing on the communication and synchronization of components. At its core are ports, which serve as connector endpoints for data transfer and synchronization, enabling the exchange of information between components. Connectors impose data and synchronization constraints on the data flow, and when all constraints are satisfied the data moves from input ports to output ports. Dual ports sharing the same name are connected forming complex circuits and linking the several components of a system.

In this paper, we use symbolic constraint automata as a semantic model of Reo connectors [12]. In symbolic constraint automata, transitions are labeled by guarded actions. Transitions may only be taken if enabled, a property expressed by a predicate on the current local state and the current values present at the input ports. In this case, an action is executed that may change the value of the local state and output ports.

For simplicity, we abstract from a concrete syntax for predicates and actions, and we denote by $\phi(\bar{x}, \bar{y}) = P(\bar{x}) \rightarrow \bar{y} := a(\bar{x})$, a guarded action, with $P(\bar{x})$ a predicate on a finite list (without repetition) $\bar{x}$ of variables in $IPrt \cup Fld$, and $a(\bar{x})$ an actions that given $\bar{x}$ as input modify the variables in $\bar{y}$, a finite list (without repetition) of variables in $OPrt \cup Fld$. The guard P is evaluated only when each input port in $\bar{x}$ receives a value from the environment (thus not equal to $\bot$). If the guard holds and all output ports in $\bar{y}$ are ready to communicate (i.e. they are all $\bot$) then the action a is executed using the values at the input ports and the current value of the fields in $\bar{x}$. The result is assigned to the variables in $\bar{y}$. Since output ports are only used to communicate a value to the environment, we assume no occurrence of them on the guard and as input of the action. Dually, since input ports receive values only from the environment, we assume no occurrence of them on the left-hand side of the assignment. We denote by $GAct(I, O, F)$ the set of all guarded action over a set $I \subseteq IPrt$, $O \subseteq OPrt$, and $F \subseteq Fld$.

We use symbolic constraint automata as introduced in [12] with additional accepting states.

Definition 4.4. A symbolic constraint automata (SCA) is a tuple $(S,\ I,\ O,\ F\ \longrightarrow,\ A,\ s_0)$ where

- S is a finite set of states;
- $I \subseteq IPrt$, $O \subseteq OPrt$, and $F \subseteq Fld$ are the relevant ports and fields;
- $\longrightarrow \subseteq S \times GAct(I, O, F) \times S$ is a transition relation;
- $A \subseteq S$ is a set of accepting states, and
- $s_0 \in S$ is a distinguished initial state.

In symbolic constraint automata, a transition denotes the possibility of executing a guarded action. However, for the actual execution of the guarded action to take place, the guard of the action must hold upon evaluation in the current assignment of variables to values.

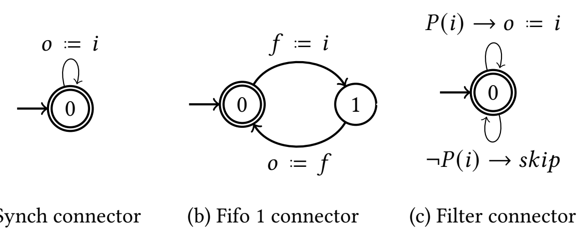

Figure 3: Three symbolic constraint automata

In Figure 3, we show three symbolic constraint automata. The one on the left corresponds to the *synchronous channel* in Reo as the data received at the input port i is synchronously passed to the output o. The symbolic constraint automaton in the middle describes Reo's *Fifo 1 channel*: it assigns to the field f the value taken from i if it is in the empty state 0, and puts to the port o the value from f if it is in the full state 1. Finally, the rightmost automaton corresponds to a *Filter channel* in Reo. If the predicate P holds when a value is available at an input port i, then the connector behaves like a synchronous connector and passes the input value to the output port o. Otherwise, $\neg P$ holds on the value of i and the value is taken from i and lost, meaning that the component waiting for synchronization on port i is released.

4.2 From SCA to ioNKA

Given a guarded action $\phi(\bar{x}, \bar{y}) = P(\bar{x}) \rightarrow \bar{y} := a(\bar{x})$ and atoms α, β assigning values to all variables (and possibly $\bot$ to some input or output ports) we denote by $P(\alpha)$ the evaluation of $P(\bar{x})$ where all occurrences of (free) variables $z \in \bar{x}$ are substituted with $\alpha(z) \in Val$. Similarly, we denote by $a(\alpha)$ the list of values obtained by evaluating a when all variables $z \in \bar{x}$ get value $\alpha(z) \in Val$. Finally, we say that the Hoare triple $\{\alpha\}\phi\{\beta\}$ holds if

- ϕ is executable under α, that is $\alpha(i) \neq \bot$ for all input ports $i \in \bar{x}$ and $\alpha(o) = \bot$ for all output port $o \in \bar{y}$.
- α is a precondition of ϕ enabling its guard, that is $\alpha \leq P(\alpha)$; and
- β is a postcondition of ϕ changing only the variables in $\bar{y}$ and consuming the value from all input ports in $\bar{x}$, that is $\alpha[a(\alpha)/\bar{y}, \bar{\bot}/\bar{i}] \leq \beta$

where $\alpha[\bar{v}/\bar{y}, \bar{\bot}/\bar{i}]$ is the atom mapping variables in $\bar{y}$ to the respective values in $\bar{v}$, enabling input ports in x to receive values, and remaining unchanged otherwise.

Pre and postconditions of a guarded action are used to construct an *ioNKA* from a symbolic constraint automaton

Definition 4.5. A SCA $(S,\ I,\ O,\ F\ \longrightarrow,\ A,\ s_0)$ can be transformed into a *ioNKA* $(S,\ Var\ \Delta,\ \Xi,\ s_0)$ with $Var = I \cup O \cup F$ and

- $s' \in \Delta(s, \alpha, \beta)$ if and only if $s \xrightarrow{\phi} s'$ and $\{\alpha\}\phi\{\beta\}$;
- $\Xi(s, \alpha, \beta)$ if and only if $s \xrightarrow{\phi} s' \in A$ and $\{\alpha\}\phi\{\beta\}$.

Here α and β are atoms in $B(Var)$.

Consider, for example, the symbolic constraint automaton in Figure 3.(b) of a Fifo 1 connector. The corresponding NetKAT automaton has the following transition and observation maps:

$$\Delta(0, \alpha, \beta) = \{1\} \qquad \Delta(1, \alpha', \beta') = \{0\}, \text{ and } \Xi(1, \alpha', \beta')$$

for any atom $\alpha \leq i = v$, $\beta \leq (i = \bot \cdot f = v)$, $\alpha' \leq (o = \bot \cdot f = u)$ and $\beta' \leq (o = u \cdot f = u)$. A string accepted by this automaton is, for example, $([i = v, o = \bot, f = u], [i = \bot, o = \bot, f = v]) \cdot ([i = \bot, o = \bot, f = v], [i = \bot, o = v, f = v])$.

As another example, the *ioNKA* obtained from the symbolic constraint automaton in Figure 3.(c) denoting a filter connector has the following transition and observation maps:

$$\Delta(0, \alpha, \beta) = \{0\} \quad \Delta(0, \alpha', \beta') = \{0\}, \text{ and } \Xi(0, \alpha, \beta) \quad \Xi(0, \alpha', \beta')$$

for any atom $\alpha \leq i = v \in P(v)$, $\beta \leq (i = \bot \cdot o = v)$, $\alpha' \leq i = v \notin P(v)$, and $\beta' \leq (i = \bot)$.

Correctness of the translation from symbolic constraint automata to NKA with respect to the following notion of bisimulation is immediate by construction. However, this bisimulation relation will become more interesting when proving the correctness of the parallel composition of two automata.

Definition 4.6. Given symbolic constraint automaton $C = (S,\ I,\ O,\ F,\ \longrightarrow,\ A,\ s_0)$ and an NKA $N = (Q,\ V\,\Delta,\ \Xi,\ q_0)$ with $V = I \cup O \cup F$, we say that a binary relation $R \subseteq S \times T$ is a bisimulation if $(s_0, q_0) \in R$ and whenever $(s, q) \in R$ then

- for all $s \xrightarrow{\phi} s'$ and $\{\alpha\}\phi\{\beta\}$ there exists $q' \in \Delta(q, \alpha, \beta)$ such that $(s', q') \in R$;
- for all $q' \in \Delta(q, \alpha, \beta)$ there exists $s \xrightarrow{\phi} s'$ such that $\{\alpha\}\phi\{\beta\}$ and $(s', q') \in R$;
- $\Xi(q, \alpha, \beta)$ holds for all $s \xrightarrow{\phi} s' \in A$ and $\{\alpha\}\phi\{\beta\}$;
- for all $\Xi(q, \alpha, \beta)$ there exists $s \xrightarrow{\phi} s'$ such that $\{\alpha\}\phi\{\beta\}$ and $s' \in A$.

Transitions with guarded actions must be matched by transitions with all pre and postconditions of those actions, and vice-versa, every pair of pre and postconditions must be related to at least one guarded action. Note that if two states q and q' of an *ioNKA* are language equivalent with respect to *IOAcc*, and a state s of an SCA is bisimilar to q then s is bisimilar to q' too, where bisimilarity is the largest bisimulation between an SCA and an *ioNKA*.

4.3 Composing ioNKA

We conclude this section with a very brief presentation of a composition operator between NetKAT automata with input and output ports inspired by the one used in Reo [4]. The idea is that the two automata synchronize via all (and only) the shared ports that are input for one automaton and output port for another. To avoid broadcasting, shared ports become local fields. No other synchronization is allowed, as all fields are only visible within the scope of an automaton. The composition is defined only if no causality problem can arise when the input and output ports of two automata are synchronized in the same step.

Definition 4.7. Let $N_1 = (S_1,\ V_1,\Delta_1,\ \Xi_1,\ s_1)$ and $N_2 = (S_2,\ V_2,\ \Delta_2,\ \Xi_2,\ s_2)$ be two non-deterministic NetKAT automata with $V_i = I_i \cup O_i \cup F_i$ for $i = 1, 2$ such that F_1 and F_2 are disjoint sets of fields in *Fld*. Assume that for every pair of (α_1, β_1) and (α_2, β_2) and state s_1 and s_2 such that either $\Delta_1(s_1, \alpha_1, \beta_1) \neq \emptyset$ and $\Delta_2(s_2, \alpha_2, \beta_2) \neq \emptyset$ or both $\Xi_1(s_1, \alpha_1, \beta_1)$ and $\Xi_2(s_2, \alpha_2, \beta_2)$ holds, the two automata synchronize only on the input ports used by one and output ports used by the other, but not on both input and output ports at the same time, that is

$$I(\alpha_1, \beta_1) \cap O(\alpha_2, \beta_2) \neq \emptyset \Rightarrow O(\alpha_1, \beta_1) \cap I(\alpha_2, \beta_2) = \emptyset\,.$$

In this case, the composition $N_1 \bowtie N_2$ is defined as the *ioNKA* $(S,\ V\,\Delta,\ \Xi,\ s_0)$ where:

- $S = S_1 \times S_2$;
- $s_0 = \langle s_1, s_2\rangle$;
- $V = I \cup O \cup F$ with $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$, $O = (O_1 \setminus I_2) \cup (O_2 \setminus I_1)$, and $F = F_1 \cup F_2 \cup (I_1 \cap O_2) \cup (I_2 \cap O_1)$;
- $\langle s', t'\rangle \in \Delta(\langle s, t\rangle, \alpha, \beta)$ whenever $s' \in \Delta_1(s, \alpha_1, \beta_1)$ and $t' \in \Delta_2(t, \alpha_2, \beta_2)$ such that if $x \in I_1 \cap O_2$ then $\alpha_1(x) = \beta_2(x) \neq \bot$, and if $x \in O_1 \cap I_2$ then $\alpha_2(x) = \beta_1(x) \neq \bot$;
- $\Xi(\langle s, t\rangle, \alpha, \beta)$ holds if both $\Xi_1(s, \alpha_1, \beta_1)$ and $\Xi_2(t, \alpha_2, \beta_2)$ hold, such that if $x \in I_1 \cap O_2$ then $\alpha_1(x) = \beta_2(x) \neq \bot$, and if $x \in O_1 \cap I_2$ then $\alpha_1(x) = \beta_2(x) \neq \bot$,

where, in the last two items, for all $i \in I, o \in O$ and $f \in F$,

$$\begin{aligned}
\alpha(i) &= \begin{cases} \alpha_1(i) & \text{if } i \in I_1 \setminus O_2 \\ \alpha_2(i) & \text{if } i \in I_2 \setminus O_1 \end{cases}\\
\beta(i) &= \begin{cases} \beta_1(i) & \text{if } i \in I_1 \setminus O_2 \\ \beta_2(i) & \text{if } i \in I_2 \setminus O_1 \end{cases}\\
\alpha(o) &= \begin{cases} \alpha_1(o) & \text{if } o \in O_1 \setminus I_2 \\ \alpha_2(o) & \text{if } o \in O_2 \setminus I_1 \end{cases}\\
\beta(o) &= \begin{cases} \beta_1(o) & \text{if } o \in O_1 \setminus I_2 \\ \beta_2(o) & \text{if } o \in O_2 \setminus I_1 \end{cases}\\
\alpha(f) &= \begin{cases} \alpha_1(f) & \text{if } f \in F_1 \cup (I_1 \cap O_2) \\ \alpha_2(f) & \text{if } f \in F_2 \cup (I_2 \cap O_1) \end{cases}\\
\beta(f) &= \begin{cases} \beta_1(f) & \text{if } o \in F_1 \cup (O_1 \cap I_2) \\ \beta_2(f) & \text{if } o \in F_2 \cup (O_2 \cap I_1) \end{cases}
\end{aligned}$$

The above operation is a congruence with respect to language equivalence as defined in Definition 4.1 and is correct with respect to the parallel operator for symbolic constraint automata as given in [12] in the sense that if there is a bisimulation relation between two symbolic constraint automata and two *ioNKA* then we can find a bisimulation between their respective parallel composition.

As an example, we show the composition of two SCA constraints automata, one representing a FIFO buffer of size taking values from the input port i, buffering in the field m_1 and outputting the buffered value at the port x, and the other similar but with input port x output port o and starting with a full buffer m_2 instead of the empty m_1.

The two symbolic constraint automata are described at the top of Figure 4, while their composition is the SCA depicted at the bottom. We concentrate on the synchronization of the transition execution of the action $m_1 := i$ with that executing the action $o := m_2$. They are implemented in the *ioNKA* in Figure 5, where $\alpha_1 = [i = v_1,\ x = v_2,\ m_1 = v_0]$, $\beta_1 = [i = \bot,\ x = v_2,\ m_1 = v_1]$, $\alpha_2 = [x = u_1,\ o = \bot,\ m_2 = u_2]$, and $\beta_2 = [x = u_1,\ o = u_2,\ m_2 = u_2]$. Here v_1 is the data received as input by the first connector and u_2 the one output by the second connector, while v_2 and u_1 are values (possibly bottom) already present at the output and input port of

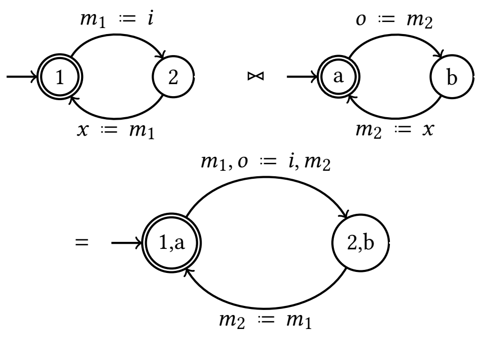

Figure 4: Symbolic constraint automata for Fifo 1 composed with Fifofull

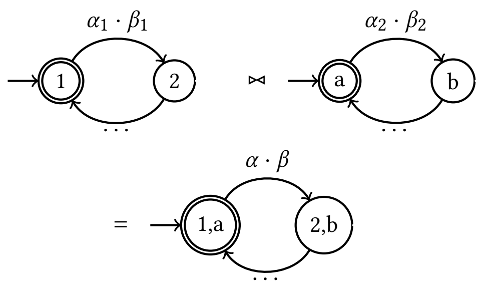

Figure 5: pNKA for Fifo 1 composed with Fifofull

the two connectors, respectively. Following the definition we get the following sets of "used" ports:

$$I(\alpha_1, \beta_1) = \{i\} \qquad O(\alpha_1, \beta_1) = \emptyset,$$
$$I(\alpha_2, \beta_2) = \emptyset \qquad O(\alpha_2, \beta_2) = \{o\}.$$

The composition of the above transitions results in the precondition $\alpha = [i = v_1,\ x = v_2,\ o = \bot,\ m_1 = v_0,\ m_2 = u_2]$, and postcondition $\beta = [i = \bot,\ x = v_2,\ o = u_2,\ m_1 = v_1,\ m_2 = u_2]$, where x becomes a local field. Note that if we create a loop and let the port $o = i$ in the second SCA then we have a problem of causality and the composition cannot take place. The problem could be solved by inserting e.g., a (synchronous) connector between o and i.

We leave it as future work the extension of the syntax of pNetKAT with an explicit declaration of input and output ports for each policy, that can be combined with the join operation $\bowtie$ as defined above.

5 CONCLUSION AND FUTURE WORK

We extended NetKAT with concurrency and communication via shared ports. We followed two semantics lines using non-deterministic constraints automata: one observing successful synchronization only, and another allowing interaction with the environment. In both cases, communication by ports played an important role, and the second one can be used as a compositional model of the Reo coordination language too.

We focussed on the operational semantics and compositionality. A possible next step is the study of axiomatizations of our two extensions. From a more practical point of view, we could use our work on model checking Reo with SPIN [12] to obtain a model checker for concurrent NetKAT. An orthogonal extension is to combine concurrency with stacks to model VLANs [25].

ACKNOWLEDGMENTS

The authors would like to thank Shuangqing Xiang for the many discussions on NetKAT automata.

REFERENCES

[1] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, Albert Rubio, Matteo Sammartino, and Alexandra Silva. 2021. Actor-based model checking for Software-Defined Networks. *Journal of Logical and Algebraic Methods in Programming* 118 (2021), 100617.

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *Acm sigplan notices* 49, 1 (2014), 113–126.

[3] Farhad Arbab. 2004. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science* 14, 3 (2004), 329–366.

[4] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. 2006. Modeling component connectors in Reo by constraint automata. *Science of computer programming* 61, 2 (2006), 75–113.

[5] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. 2014. Vericon: towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation.* 282–293.

[6] Gerd Behrmann, Alexandre David, and Kim G Larsen. 2004. A tutorial on uppaal. *Formal methods for the design of real-time systems* (2004), 200–236.

[7] Stephen Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Information and Computation* 127, 2 (1996), 145–163.

[8] Georgiana Caltais, Hossein Hojjat, Mohammad Reza Mousavi, and Hünkar Can Tunç. 2022. DyNetKAT: An Algebra of Dynamic Networks.. In *FoSSaCS*. 184–204.

[9] Dexian Chang, Ning Zhu, and Yingjie Yang. 2021. Security Analysis of SDN Access Control Protocol Based on ProVerif. In *Proceedings of IEEE 3rd International Conference on Civil Aviation Safety and Information Technology (ICCASIT).* 1155–1159.

[10] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. 2016. SDNRacer: Concurrency Analysis for Software-Defined Networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* Association for Computing Machinery, New York, NY, USA, 402–415.

[11] Hui Feng, Farhad Arbab, and Marcello Bonsangue. 2019. A Reo model of software defined networks. In *International Conference on Formal Engineering Methods.* Springer, 69–85.

[12] Hui Feng, Marcello Bonsangue, and Benjamin Lion. 2022. From symbolic constraint automata to Promela. *Journal of Logical and Algebraic Methods in Programming* 128 (2022), 100794.

[13] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A coalgebraic decision procedure for NetKAT. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 343–355.

[14] Paul Göransson and Chuck Black. 2016. *Software Defined Networks: A Comprehensive Approach.* Morgan Kaufmann.

[15] Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

[16] S-STQ Jongmans, Tobias Kappé, and Farhad Arbab. 2017. Constraint automata with memory cells and their composition. *Science of Computer Programming* 146 (2017), 50–86.

[17] Sung-Shik TQ Jongmans and Farhad Arbab. 2012. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science* 22, 1 (2012).

[18] Miyoung Kang, Eun-Young Kang, Dae-Yon Hwang, Beom-Jin Kim, Ki-Hyuk Nam, Myung-Ki Shin, and Jin-Young Choi. 2013. Formal Modeling and Verification of SDN-OpenFlow. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation.* 481–482.

[19] Dexter Kozen. 2014. NetKAT — A Formal System for the Verification of Networks. In *Programming Languages and Systems*, Jacques Garrigue (Ed.). Springer, 1–18.

[20] Nancy Lynch. 1998. I/O automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems.* Princeton University, Princeton, N.J, 29–38.

[21] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. 2016. Event-driven network programming. *ACM SIGPLAN Notices* 51, 6 (2016), 369–385.

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review* 38, 2 (2008), 69–74.

[23] Cole Schlesinger, Michael Greenberg, and David Walker. 2014. Concurrent NetCore: From policies to pipelines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming.* 11–24.

[24] Jana Wagemaker, Nate Foster, Tobias Kappé, Dexter Kozen, Jurriaan Rot, and Alexandra Silva. 2022. Concurrent NetKAT. In *European Symposium on Programming.* Springer, Cham, 575–602.

[25] Shuangqing Xiang, Marcello Bonsangue, and Huibiao Zhu. 2019. PDNet: A Programming Language for Software-Defined Networks with VLAN. In *International Conference on Formal Engineering Methods.* Springer, 203–218.

[26] Shuangqing Xiang, Huibiao Zhu, Xi Wu, Lili Xiao, Marcello M. Bonsangue, Wanling Xie, and Lei Zhang. 2020. Modeling and verifying the topology discovery mechanism of OpenFlow controllers in software-defined networks using process algebra. *Sci. Comput. Program.* 187 (2020), 102343.