

# Tracer: A Tool for Race Detection in Software Defined Network Models

Georgiana Caltais

University of Twente  
The Netherlands

`g.g.c.caltais@utwente.nl`

Mahboobeh Zangiabady

University of Twente  
The Netherlands

`m.zangiabady@utwente.nl`

Ervin Zvirbulis

University of Twente  
The Netherlands

`e.zvirbulis@student.utwente.nl`

Software Defined Networking (SDN) has become a new paradigm in computer networking, introducing a decoupled architecture that separates the network into the data plane and the control plane. The control plane acts as the centralized brain, managing configuration updates and network management tasks, while the data plane handles traffic based on the configurations provided by the control plane. Given its asynchronous distributed nature, SDN can experience data races due to message passing between the control and data planes. This paper presents Tracer, a tool designed to automatically detect and explain the occurrence of data races in DyNetKAT SDN models. DyNetKAT is a formal framework for modeling and analyzing SDN behaviors, with robust operational semantics and a complete axiomatization implemented in Maude. Built on NetKAT, a language leveraging Kleene Algebra with Tests to express data plane forwarding behavior, DyNetKAT extends these capabilities by adding primitives for communication between the control and data planes. Tracer exploits the DyNetKAT axiomatization and enables race detection in SDNs based on Lamport vector clocks. Tracer is a publicly available tool.

## 1 Introduction

Traditional network devices have been called “the last bastion of mainframe computing” [7]. Since the 1970s, network design principles have remained fundamentally unchanged, maintaining their core structure for nearly four decades. One of such fundamentals is the handling of the data and control planes. Intuitively, the data plane is a distinct functional layer in networking responsible for the forwarding of data packets between network devices. The control plane is another layer responsible for network control including policy enforcing and routing configuration. In a traditional network, each switch autonomously manages its interpretation of the control plane as illustrated in Figure 1. This architectural rigidity increases complexity in network maintainability due to the necessity of configuring each switch individually.

In response, the concept of software-defined networking (SDN) has emerged. The main difference is the separation of the data and control planes and consolidation of the management over the control plane in a centralized location as illustrated in Figure 2. SDN architectures comprise central controllers and programmable switches that communicate via standardized protocols. The former respond to network events such as new connections from hosts, topology changes, and shifts in traffic load, by reprogramming the switches accordingly (as indicated by the orange dotted arrows in the figure). Such an approach enhances network controllability and adaptability in real-time scenarios. SDN is being adopted across various leading tech companies and cloud service providers to enhance the agility, efficiency, and scalability of their data center networks. Examples include Google, Microsoft, Facebook and Amazon, just to name a few.

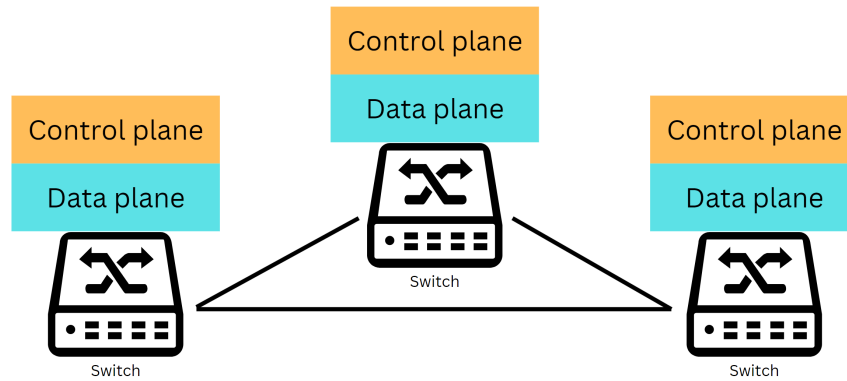


Figure 1: Traditional network setup

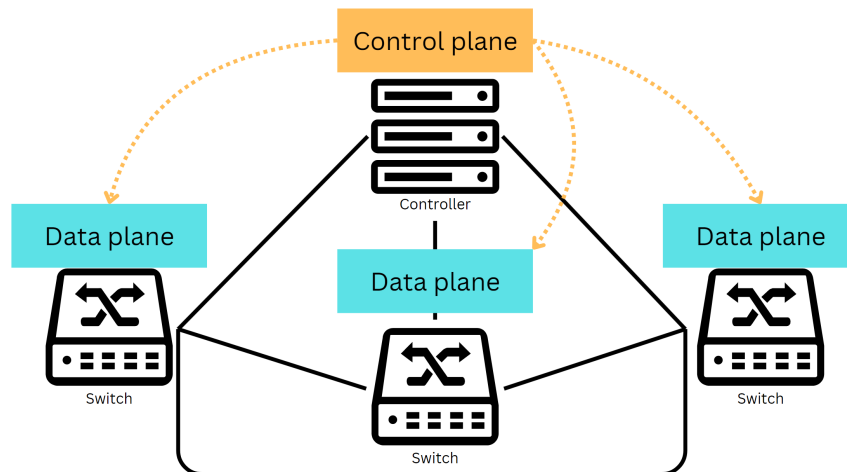
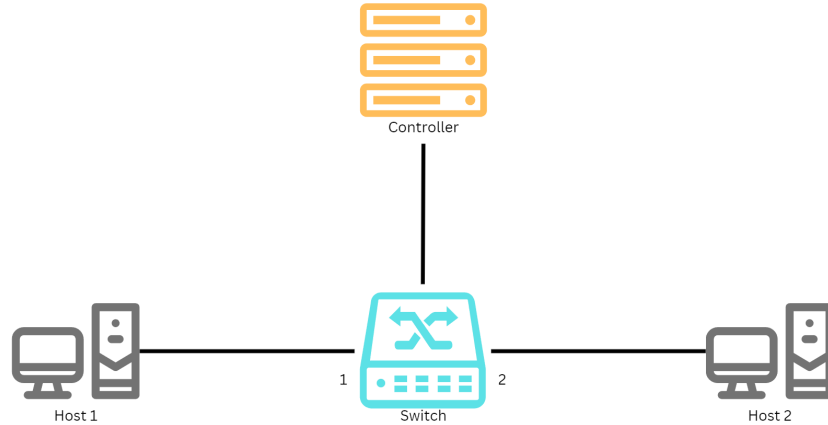


Figure 2: Software-defined network setup

SDNs are highly concurrent systems, designed to handle numerous simultaneous operations. Consequently, they are prone to data races. The latter can lead to undesired outcomes/behaviours of the SDN, especially if the races correspond to concurrency between the data and the control planes.

Let us consider Figure 3 (inspired from [4]), illustrating a basic example of an SDN consisting of: (i) a switch with two ports (1 and 2), (ii) one controller communicating with the switch, and (iii) two hosts (Host 1 and Host 2) that can send/receive packets to/from the switch via the aforementioned ports. Assume the following over-simplified scenario: The switch is configured to allow any traffic from port 1 to port 2. When the switch encounters a “blocking” flag, it notifies the controller and continues forwarding subsequent packets until a new forwarding policy (“drop everything”, in this case) is received from the controller. If Host 1 sends a packet flagged “blocking” to the switch, a data race may occur. The race arises because the outcome for a new packet depends on the timing. The new packet will either be forwarded according to the existing forwarding policy installed in the switch, if it arrives before the blocking rule from the controller, or it will be dropped if the blocking rule is in place first.

In this paper, we propose Tracer [1], a tool for the automated detection of data races in SDNs. Tracer builds around DyNetKAT [5], a formal framework for the rigorous modelling and analysis of SDNs. DyNetKAT can encode and simulate packet forwarding within SDNs, together with the actual communication between the control and data planes (or, dynamic network reconfigurations). The DyNetKAT



**Figure 3:** Running example inspired from [10]

language is supported by a rigorous operational semantics and a sound and complete axiomatisation enabling reasoning about equivalence of DyNetKAT programs, and associated packet forwarding behaviour in a fully automated fashion. Intuitively, Tracer takes as input DyNetKAT models and checks whether data races between the control and data planes occur, by following the symbolic approach in [4]. Furthermore, Tracer provides explanations of how such races can be enabled via (minimal) sequences of packets fed to the network. Such explanations can serve as a great debugging aid for the network administrators.

Several methods have been developed to detect race conditions in SDNs, including ConGuard [13] and SDNRacer [6]. These tools identify race condition vulnerabilities by analyzing dynamically generated log files to construct an event graph where happens-before edges connect events, and race conditions manifest as partially ordered events. The Spin model checker, as discussed in [12], has also been employed to detect race conditions through runtime monitoring of events in SDNs. In contrast, this paper is based on a *static approach to identify races* in SDNs, eliminating the need for dynamic log generation from a network.

*Our contributions:* As previously mentioned, in this paper we introduce Tracer [1], a tool for the automated detection of races in SDN models encoded in DyNetKAT, based on the theoretical framework in [4]. In short, Tracer exploits the symbolic semantics of DyNetKAT in [4] and uses Lamport vector clocks for detecting races entailed by the concurrent message passing between the SDN control and data planes. Furthermore, Tracer provides explanations of how such races are enabled by computing minimal sets of network packets that lead to not well-behaved communication scenarios. The instructions for installing and running Tracer are publicly available in [1].

*Structure of paper:* In Section 2, we briefly recall (Dy)NetKAT and introduce our running example. In Section 3, we present the idea behind vector clocks for race detection in distributed systems. The symbolic semantics of DyNetKAT enriched with vector clocks is recalled in Section 4. Our tool, Tracer, is introduced in Section 5. We draw the conclusions and provide pointers to future work in Section 6.

## 2 Overview of DyNetKAT

DyNetKAT [5] serves as a framework for representing and analyzing the behaviors of SDNs, such as packet forwarding and the interaction between the control and data planes. DyNetKAT is an extension

of NetKAT [2], which is a language based on Kleene Algebra with Tests [8], tailored for modeling and analyzing data plane forwarding. DyNetKAT introduces concurrency to NetKAT, in order to support dynamic reconfigurations of the data plane, such as the installation of new forwarding rules, in line with control plane protocols.

Figure 4 illustrates the syntax and semantics of the NetKAT language. Network packets are encoded in (Dy)NetKAT as collections of fields, and associated values ranging over finite domains:  $\{f_1 = v_1, \dots, f_n = v_n\}$ . For instance, a packet  $\sigma$  of type *SSH* residing at port 1 of switch  $SW_A$ , with destination  $Host_1$ , can be conveniently denoted as  $\sigma \triangleq \{type = SSH, pt = 1, SW = SW_A, dst = Host_1\}$ . The main syntactic elements of NetKAT include primitives ( $Pr$ ) for dropping incoming packets ( $\mathbf{0}$ ) and accepting incoming packets without further processing ( $\mathbf{1}$ ). NetKAT primitives can filter out packets based on tests ( $f = n$ ) and their disjunction ( $+$ ), conjunction ( $\cdot$ ) and negation ( $\neg$ ). NetKAT policies ( $N$ ) can also be used for packet fields modifications ( $f \leftarrow n$ ), or to express packet multicasting ( $+$ ), composition of policies ( $\cdot$ ) and iteration ( $*$ ). The operator **dup** is designed for building histories of packets processed by an SDN dataplane encoded in NetKAT. The denotational semantics of NetKAT is defined over sets of packet histories as in Figure 4; an intuitive description of its operators has been provided earlier. Furthermore, NetKAT has a sound and complete axiomatization that has been effectively used to reason about packet reachability within NetKAT models.

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

**Figure 4:** NetKAT: Syntax and Semantics [2]

The syntax of DyNetKAT is defined on top of the **dup**-free fragment of NetKAT as in (1). The constant  $\perp$  denotes a DyNetKAT process without behaviour. Sequential composition of DyNetKAT policies  $D$  is denoted by  $;$ . The operator  $\parallel$  encodes concurrent behaviours of DyNetKAT policies, whereas  $\oplus$  stands for non-deterministic choice. (A)synchronous communication in DyNetKAT is modeled in an ACP [3]-style via message sending operators  $x!N;D$  and receiving operators  $x?N;D$ . Intuitively, messages  $N$  (e.g., NetKAT flow tables) can be exchanged via channels  $x$  as a result of the communication between the control and data planes. As soon as such a new forwarding policy  $N$  is received via  $x$ , the continuation  $D$  can update its behaviour according to  $N$ . This would correspond to installing a new forwarding policy  $N$  in the dataplane. Variables  $X$  enable defining recursive DyNetKAT policies.

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

The operational semantics of DyNetKAT is given in Fig. 5, over tuples of shape  $(D, H, H')$ , where  $D$  is a DyNetKAT policy,  $H$  is the list of packets waiting to be processed by the network, and  $H'$  is the

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

**Figure 5:** DyNetKAT: Operational Semantics (relevant excerpt)

history of packets being processed according to the forwarding rules in the data plane. Rule  $(\mathbf{cpol}'_{\cdot})$  in Fig. 5, for instance, processes the current packet  $\sigma$  (at the top of the waiting list) according to the NetKAT flow table encoded by  $p$ . The possibly modified packet is  $\sigma'$ , and a corresponding transition  $\xrightarrow{(\sigma, \sigma')}$  can be observed in the behaviour Labelled Transition System (LTS) of the DyNetKAT model.  $\sigma'$  is added to the history  $H'$ , and the execution of the model proceeds with the continuation  $q$  and the remaining waiting packets in  $H$ . Rule  $(\mathbf{cpol}_{\clubsuit, \spadesuit})$ , for instance, encodes synchronous communication in DyNetKAT: a new forwarding rule or NetKAT policy  $p$  is communicated via channel  $x$  in a handshake between two parallel SDN components  $q \parallel s$  (e.g., one controller  $q$  and one switch  $s$ ). The handshake entails an execution  $\xrightarrow{\mathbf{rcfg}(x, p)}$  within the DyNetKAT model. Rules  $(\mathbf{cpol}_{\oplus})$  and  $(\mathbf{cpol}_{\parallel})$  and their symmetric counterparts define non-deterministic choice and parallel composition, respectively, in a standard fashion. Rule  $(\mathbf{cpol}_x)$  simply replaces recursive variables with their definitions. Rule  $(\mathbf{cpol}_{\bullet})$  encodes the axioms for asynchronous communication. Furthermore, DyNetKAT has an ACP-like sound and complete axiomatisation for LTS bisimilarity. A complete and thorough presentation of the DyNetKAT formal framework can be found in [5].

## 2.1 Running Example

Next, we illustrate the DyNetKAT framework by means of an example. Consider the scenario in Figure 3. A possible encoding in DyNetKAT is given in (2), as follows.

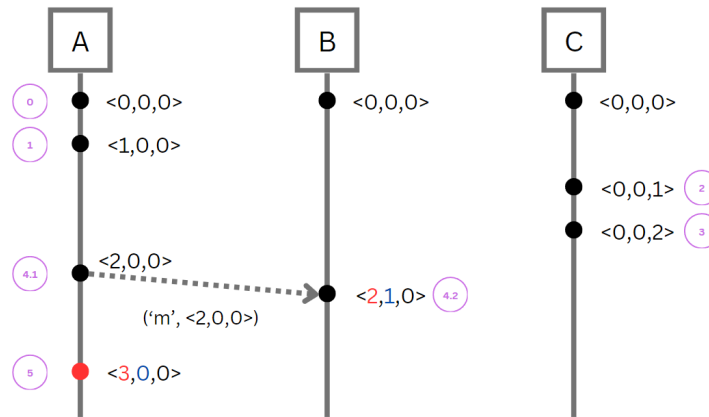
$$\begin{aligned}
 SW &\triangleq (flag = regular) \cdot (pt = 1) \cdot (pt \leftarrow 2); SW \oplus \\
 &\quad (flag = blocking) \cdot (pt = 1); ((Help ! 1); SW) \oplus \\
 &\quad (Up ? 1); SW' \\
 SW' &\triangleq \mathbf{0}; \perp \\
 C &\triangleq (Help ? 1); ((Up ! 1); C)
 \end{aligned} \tag{2}$$

We write  $(flag = regular)$  for packets **not** of the blocking type. Whenever such a packet arrives at

port 1 ( $pt = 1$ ) of the switch ( $SW$ ), it gets forwarded to port 2 ( $pt \leftarrow 2$ ). Then, the switch continues recursively (denoted by  $;SW$ ). Alternatively (denoted by  $\oplus$ ), we write ( $flag = blocking$ ) to encode matching of packets of blocking type. Whenever such a packet arrives at port 1, the switch informs the controller that a new forwarding rule needs to be installed (denoted by sending the message  $Help!1$ ). The new blocking behaviour is announced to the switch via  $Up!1$ . Upon receiving the message  $Up!1$ , the forwarding table of  $SW$  is updated to  $SW'$ . The latter drops any incoming packet ( $\mathbf{0}$ ) and irreversibly stops from processing packets ( $\perp$ ). The controller ( $C$ ) repeatedly listens on channel  $Help?$  for requests from the switch, and instructs the switch to install the blocking behaviour  $SW'$  via  $Up!1$ .

### 3 Vector Clocks

SDN is a paradigm that falls under the definition of a distributed system [11]. In this case, the components are controllers and switches, and the whole network represents a distributed system. In distributed systems like SDNs, a data race means that switches and controllers perform actions concurrently, possibly leading to undesired behaviours. As illustrated in the introduction, for instance, there might be the case that due to concurrency, the network still forwards unsafe packets in between a new forwarding policy request, and the actual installation of the new forwarding rules. We call these *data races between the control and data planes*. One possible approach to detecting data races is the use of vector clocks [9]. Each such clock is associated with a component in the distributed system, and it consists of a vector of size equal with the number of components in the system. Each vector entry in a clock counts actions performed by a distinct component. (In the context of DyNetKAT models, for instance, actions stand for packet forwarding or reconfigurations between the control and data planes.) Each component in the system has its own copy of a vector clock as illustrated in Figure 6.



**Figure 6:** Vector clocks in a system with three parallel components A, B and C

As shown in Figure 6, in step 0 all three clocks are initialized with zeros. When a component performs an individual action (i.e., no message-passing involved), it increments its own index in its local copy of the vector clock. The rest of the entries in the clock, as well as the clocks of the other components, are unchanged (see, e.g., step ① or ⑤ of component A). Both steps ② and ③ correspond to a similar scenario, but within component C.

Synchronous communication is handled as follows: Once a component sends a message, it first increments its clock, and then sends it along with the message, creating a timestamped message (step ④.1).

Upon its arrival, the receiver updates the rest of the clock entries in the local copy if the corresponding entries in the message timestamp are greater, and then it increments its clock entry (step ④.2). Note that step 4 consists of two parts capturing the synchronous sending and receiving of a message in one time frame (i.e., captures a handshake communication).

To understand how vector clocks help detect data races we first need to know what does it mean for two vector clocks to be comparable. Consider two vector clocks  $V_i$  and  $V_j$  of size  $k$ . The clocks are comparable if:

$$V_i[x] \leq V_j[x] \quad \forall x \in \{1, \dots, k\}$$

or

$$V_i[x] \geq V_j[x] \quad \forall x \in \{1, \dots, k\}$$

Such comparable pair of vector clocks indicates that the associated components did not run concurrently. If there exist  $x, y \in \{1, \dots, k\}$  with  $x \neq y$  such that:

$$(V_i[x] \not\leq V_j[x] \quad \wedge \quad V_i[y] \not\geq V_j[y])$$

then we conclude that components associated with  $V_i$  and  $V_j$  operate concurrently, implying a data race. Steps ⑤ and ④.2 for instance, witness concurrent behaviour between A and B. Similarly for ⑤ and ②, etc.

## 4 Overview of Symbolic DyNetKAT

In this section, we briefly recall the contribution in [4] that introduces a symbolic operational semantics of DyNetKAT, enriched with vector clocks for detecting races between the control and data planes. The most important idea behind the symbolic DyNetKAT reduces to exploiting the so-called DyNetKAT *head normal forms* that enable simulating packet processing within SDN models in a purely syntactic fashion, without actual packets being “fed” to the network.

The idea is as follows: each (guarded) DyNetKAT policy  $d$  can be equivalently expressed (based on its complete axiomatisation in [5]) as a sum ( $\oplus$ ) of DyNetKAT policies of shape  $\alpha \cdot \pi; d'$  or  $\mathbf{rcfg}(\mathbf{x}, \mathbf{n}); d'$ . Here,  $\alpha$  stands for a so-called *complete test*  $(f_1 = v_1) \cdot \dots \cdot (f_n = v_n)$  encoding all the conditions an incoming packet has to match within a flow table, in order to be forwarded accordingly. Each packet passing a complete test as before is, in fact, a packet of shape  $\sigma_\alpha \triangleq \{f_1 = v_1, \dots, f_n = v_n\}$ ; so, a complete test encodes an incoming packet. A *complete assignment*  $\pi$  as before, is a policy  $(f_1 \leftarrow v'_1) \cdot \dots \cdot (f_n \leftarrow v'_n)$  encoding how the packet matching the complete test is processed by the data plane. Basically, a complete assignment encodes a forwarded/processed packet  $\sigma_\pi \triangleq \{f_1 = v'_1, \dots, f_n = v'_n\}$ . It is, therefore, easy to understand that the symbolic semantics of DyNetKAT can be defined based on such normal forms which entail transitions of shape  $\xrightarrow{(\sigma_\alpha, \sigma_\pi)}$  and  $\xrightarrow{\mathbf{rcfg}(\mathbf{x}, \mathbf{n})}$ , respectively, without the need of actual packets.

In [4], each SDN encoding a set of parallel switches ( $S_i$ ) and controllers ( $C_j$ )

$$S_1 \parallel \dots \parallel S_n \parallel C_1 \parallel \dots \parallel C_m \tag{3}$$

is enriched with vector clocks  $\vec{c}_k$  associated with each component

$$S_1 \vec{c}_1 \parallel \dots \parallel C_m \vec{c}_m \tag{4}$$

entailing DyNetKAT symbolic operational rules. For instance:

$$(\mathbf{Symb}_{\checkmark}) \frac{p_i \in \text{NetKAT}^{-\text{dup}} \quad n.f.(p_i) = \sum_{\alpha_i, \pi_i \in \mathcal{A}} \alpha_i \cdot \pi_i}{(p_i; q_i)_{\vec{c}_i} \parallel \prod_{\substack{1 \leq j \leq n \\ j \neq i}} d_j \vec{c}_j \xrightarrow{(\sigma_{\alpha_i}, \sigma_{\pi_i})} (q_i)_{\vec{c}_i[i]++} \parallel \prod_{\substack{1 \leq j \leq n \\ j \neq i}} d_j \vec{c}_j}$$

is the symbolic counterpart of  $(\mathbf{cpol}_{\checkmark})$ , where the vector clock  $\vec{c}_i$  of the “evolving” component  $p_i; q_i$  is incremented in accordance with the semantics of the vector clocks in Section 3, and the input packet  $\sigma_{\alpha_i}$  and the processed packet  $\sigma_{\pi_i}$  defining this step  $\xrightarrow{(\sigma_{\alpha_i}, \sigma_{\pi_i})}$  are entailed based on the normal form of  $p_i$  (note that normal forms exist for NetKAT as well [2]).

Here we write:

$$P_{i\vec{c}_i} \parallel \prod_{\substack{1 \leq j \leq k \\ j \neq i}} P_{j\vec{c}_j}$$

to denote

$$P_{1\vec{c}_1} \parallel \dots \parallel P_{k\vec{c}_k}$$

The symbolic rule for handshake (i.e., the counterpart of  $(\mathbf{cpol}_{\parallel})$ ) is defined in a similar fashion, where both vector clocks of the communicating SDN components are updated, and the transition step is marked as  $\xrightarrow{\mathbf{rfg}(\mathbf{x}, \mathbf{p})}$ :

$$(\mathbf{Symb}_{\parallel}) \frac{h.n.f(q_i) \triangleq x!q; d_i \oplus r_i \quad h.n.f(q_k) \triangleq x?q; d_k \oplus r_k}{(q_1)_{\vec{c}_1} \parallel \dots \parallel (q_i)_{\vec{c}_i} \parallel \dots \parallel (q_k)_{\vec{c}_k} \parallel \dots \parallel (q_n)_{\vec{c}_n} \xrightarrow{\mathbf{rfg}(\mathbf{x}, \mathbf{p})} (q_1)_{\vec{c}_1} \parallel \dots \parallel (d_i)_{\vec{c}_i[i]++} \parallel \dots \parallel (d_k)_{(\max(\vec{c}_i[i]++, \vec{c}_k))[k]++} \parallel \dots \parallel (q_n)_{\vec{c}_n}}$$

Figure 7 illustrates the symbolic execution of the SDN in (2). For brevity of notation, we write:  $\sigma_{B,1}$  to denote a packet  $\{flag = blocking, pt = 1\}$ ,  $\sigma_{R,1}$  to encode a packet  $\{flag = regular, pt = 1\}$  and  $\sigma_{R,2}$  in lieu of  $\{flag = regular, pt = 2\}$ . As intuitively explained in Section 1: if Host 1 starts sending blocking traffic to the switch on port 1, a data race may occur. The race arises because the outcome for a new packet depends on the timing. The new packet will either be (i) forwarded according to the existing forwarding policy installed in the switch, if it arrives before the blocking rule from the controller, or (ii) it will be dropped if the blocking rule is in place first. Case (i) matches the symbolic execution  $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow n_4$ : instead of immediately installing the “drop everything” policy in  $SW'$ , the network first forwards a regular packet from port 1 to port 2. The race is detected by the incomparable clocks  $\langle 3, 0 \rangle$  and  $\langle 2, 1 \rangle$  in  $n_4$ . Furthermore, the sequence associated packets  $\sigma_{B,1}$  and  $\sigma_{R,1}$  can be seen as the (minimal) explanation of the race. A similar reasoning holds for the race in  $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow n_6$ . Case (ii) corresponds to the symbolic execution  $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow n_3$ . Note that all vector clocks can be compared along this execution, so no data race is identified.

## 5 Tracer

Tracer [1] is the tool developed in this work. It exploits the symbolic semantics of DyNetKAT as described in Section 4, and computes minimal sets of packets that enable races between the control and



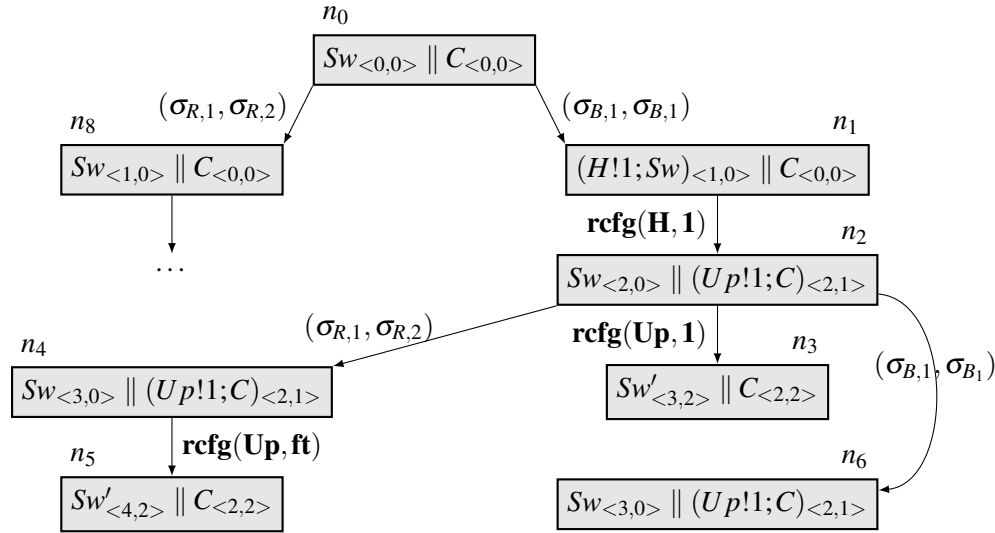


Figure 7: Symbolic execution of the SDN in (2); excerpt

data planes of an inputted SDN encoded in DyNetKAT. In this section we provide the algorithm behind Tracer, instructions on how to install and run the tool, and one example of using Tracer.

We define a race detection function  $rd(SDN, k)$  in (5) that identifies minimal symbolic executions of SDN witnessing races up to a given depth  $k$  in the execution tree of SDN (as illustrated in Section 4). Furthermore, the function returns the network packets enabling these races, as explanations. The traced packets are encoded as  $\alpha^i$  in (5).(d). Recall that every complete test  $\alpha_i = (f_1 = v_1) \cdot \dots \cdot (f_m = v_m)$  entails a unique packet  $\{f_1 = v_1, \dots, f_m = v_m\}$ . The Tracer Algorithm 1 implements the  $rd(-)$  function based on an interplay behind Python and Maude. Invoking Maude is for deriving DyNetKAT policies in head normal forms according to the DyNetKAT complete axiomatization in [5]. These head normal forms (denoted by  $hnf(d_i)$  and  $hnf(d_j)$  in (5).(d) and in (5).(e)) are further exploited for identifying packet forwarding steps within the analysed SDN ( $\alpha^i \cdot \pi_{\alpha^i}$  in (5).(d)), or communication steps between the data and control planes ( $\mathbf{rcfg}(\mathbf{X}, \mathbf{p})$  based on  $X_{\gamma^i}!p_{\gamma^i}$  and  $X_{\gamma^j}?p_{\gamma^j}$  in (5).(e)). Observe that vector clocks are updated in accordance with the clocks of the symbolic rules: (5).(d) complies to  $(\mathbf{Symb}_{\checkmark})$  and (5).(e) complies to  $(\mathbf{Symb}_{\parallel})$ , respectively. The function  $not\text{-}race(\vec{C}_1, \dots, \vec{C}_n)$  returns **true** whenever any two vector clocks  $\vec{C}_i$  and  $\vec{C}_j$  are incomparable, and **false** otherwise.

$$\begin{aligned}
 (a) \quad & rd(\prod_{1 \leq i \leq n} d_{i\vec{c}_i}, 0) \triangleq \emptyset \\
 (b) \quad & rd(\prod_{1 \leq i \leq n} d_{i\vec{c}_i}, k+1) \triangleq \\
 (c) \quad & \text{if } not\text{-}race(\vec{C}_1, \dots, \vec{C}_n) \text{ then return} \\
 (d) \quad & \bigcup_{\substack{1 \leq i \leq n \\ (\alpha^i \in A^i) \wedge (\alpha^i \cdot \pi_{\alpha^i}; d_{\alpha^i} \in hnf(d_i))}} \alpha^i :: rd(d_{\alpha^i \vec{c}_i[\vec{i}]+} \parallel \prod_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\vec{c}_j}, k) \\
 (e) \quad & \bigcup_{\substack{1 \leq i \neq j \leq n \\ (\gamma^i \in \Gamma^i) \wedge (X_{\gamma^i}!p_{\gamma^i}; d_{\gamma^i} \in hnf(d_i)) \\ (\gamma^j \in \Gamma^j) \wedge (X_{\gamma^j}?p_{\gamma^j}; d_{\gamma^j} \in hnf(d_j)) \\ (X_{\gamma^i} == X_{\gamma^j} == X) \wedge (p_{\gamma^i} == p_{\gamma^j} == p)}} \mathbf{rcfg}(\mathbf{X}, \mathbf{p}) :: rd(d_{\gamma^i \vec{c}_i[\vec{i}]+} \parallel d_{\gamma^j \max(\vec{c}_i[\vec{i}]+, \vec{c}_j[\vec{j}]+)} \parallel \prod_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\vec{c}_j}, k) \\
 (f) \quad & \text{else return } \downarrow
 \end{aligned} \tag{5}$$

We use  $\downarrow$  in (5).(f) as a marker symbol indicating that  $rd(-)$  identified a race witnessing trace. Every

trace ending with  $\downarrow$  returned by  $rd(SDN, k)$  encodes a set of packets witnessing concurrent behaviour within the SDN. We use  $::$  in (5) as a constructor (concatenation) for such witnesses.

From an algorithmic perspective: (5).(d) is handled in lines 13 – 21 of Algorithm 1, whereas (5).(e) is handled in lines 22 – 31. Note that the aforementioned head normal forms in (5) are computed using the DyNetKAT axiomatization implemented in Maude [5]: lines 8, 15, 24 and 25 in Algorithm 1. Line 8 invokes the application of a Maude-defined “projection” operator  $\text{pi}\{m\}$  that unfolds the given expression  $N_i$  up to depth  $m$ . Checking for deadlock in line 12 of Algorithm 1 is a stopping condition based on whether all parallel components in  $\text{curr}$  are either  $\perp$  or start with communication actions that cannot be matched by any other component. Lines 33 – 34 extract the race witnessing packets, in a post-processing step.

**Tracer at Work.** Tracer is publicly available at [1]. A complete installation guide can be found in README.md. Requirements for running Tracer include a Linux operating system, specifically Ubuntu 20.04<sup>1</sup> with Python (version  $> 3.10.12$ ). The tool also uses Maude 3.1, that is included in the installation of Tracer. To use it, run the command in the following form:

```
> python tracer\_runner.py <path_to_maude> <path_to_model_in_maude>
```

The command has several optional parameters as given in Table 1. Note that the parameters with values should be inputted without the space between the parameter and the value. For example, write `-grace` to produce only graphs and traces witnessing data races in the provided model.

**Table 1:** Tracer command line

Parameter	Value	Explanation
-c	-	output text with color
-t	-	show tracing steps
-u	int	unfold depth
-g	‘race’ or ‘full’	types of trees and traces to generate (race witnesses only, or full trees/traces)
-f	string	set a name for text output file (copy of console output)

**SDN Encoding in Tracer: Example.** The DyNetKAT encoding in (2) is provided as input for Tracer in a Maude-compatible format as shown in Listing 1. The DyNetKAT recursive variables  $SW, SW'$  and  $C$  in (2) are declared as the constants  $SW, SWP$  and  $C$  of `Recursive` type in Listing 1. The operator `getRecPol(...)` is a syntactic wrapper around these recursive operators. The actual definitions of the switch and controller follow closely the syntax in (2). The communication channels  $Help$  and  $Up$  translate to the constants `Help` and `Up` of type `Channel` in Maude. The DyNetKAT non-deterministic choice  $\oplus$  translates to `o+` in Maude. The DyNetKAT constants  $\mathbf{0}$  and  $\perp$  are mapped to `zero` and `bot`. The entire SDN consisting of the switch  $SW$  and controller  $C$  as in Figure 3 is defined by the constant `Init` of type `DNA` in Maude. Note that the NetKAT expressions encoding the forwarding policies are provided as strings in Maude; e.g., `"(flag = regular).(pt = 1).(pt <- 2)"`. The model in Listing 1 along with the depth  $k$  of the analysis are provided as input to Tracer.

<sup>1</sup>Other Linux distributions might work, however, the development and testing were done on the specified version of Ubuntu.

```

1: Input: SDN with  $k$  components as a DyNetKAT model  $(N_1 \parallel N_2 \parallel \dots \parallel N_k)$ 
2: Input: Depth  $m$  of the search
3: Output: The smallest sets of network packets enabling races in the SDN
4: for  $i \in \{1, \dots, k\}$  do
5:   Initialize vector clocks  $C_i = \langle 0, \dots, 0 \rangle$  of size  $k$ 
6: end for
7: for  $i \in \{1, \dots, k\}$  do
8:   Let  $pmN_i$  be the result of invoking Maude > reduce pi{m}(N_i)
9: end for
10: Initialize curr with  $(pmN_1, C_1) \parallel (pmN_2, C_2) \parallel \dots \parallel (pmN_k, C_k)$ 
11: Initialize symb-traces with  $\emptyset$ 
12: if not-deadlock(curr) then
13:   for each element  $(N_i, C_i)$  at position  $i$  in curr do
14:     if (complete-test-assignment;  $d_i$ ) is a summand of  $N_i$  then
15:       Let  $rd_i$  be the result of invoking Maude > reduce d_i
16:       Set curr to  $(N_1, C_1) \parallel \dots \parallel (rd_i, C'_i) \parallel \dots \parallel (N_k, C_k)$ 
17:       where  $C'_i$  is  $C_i$  incremented at position  $i$ 
18:       Append (complete-test,  $(C_1, \dots, C'_i, \dots, C_k)$ ) to symb-traces
19:       go to step 12
20:     end if
21:   end for
22:   for all pairs of elements  $(N_i, C_i)$  and  $(N_j, C_j)$  at positions  $i$  and  $j$  in curr do
23:     if  $(X!p; d_i)$  is a summand of  $C_i$  and  $(X?p; d_j)$  is a summand of  $C_j$  then
24:       Let  $rd_i$  be the result of invoking Maude > reduce d_i
25:       Let  $rd_j$  be the result of invoking Maude > reduce d_j
26:       Set curr to  $(N_1, C_1) \parallel \dots \parallel (rd_i, C'_i) \parallel \dots \parallel (rd_j, C'_j) \parallel \dots \parallel (N_k, C_k)$ 
27:       where  $C'_i$  is  $C_i$  incremented at position  $i$ , and  $C'_j$  is  $\max(C'_i, C_j)$  incremented at position  $j$ 
28:       Append (rcfg( $X, p$ ),  $(C_1, \dots, C'_i, \dots, C'_j, \dots, C_k)$ ) to symb-traces
29:       go to step 12
30:     end if
31:   end for
32: end if
33: Let races be a set of sets of packets, initialized with  $\emptyset$ 
34: for all symb-trace in symb-traces do
35:   if a prefix s-tr' of symb-trace ends with incomparable vector clocks  $(C_1, \dots, C_k)$  then
36:     Extract all packets pkt based on every max-test in s-tr'
37:     Add the set of packets pkt to races
38:   end if
39: end for
40: return races

```

**Algorithm 1:** Detecting races in SDN using DyNetKAT models

```

fmod MODEL is
[...]

ops Init : -> DNA .
ops SW, SWP, C : -> Recursive .
ops Help, Up : -> Channel .

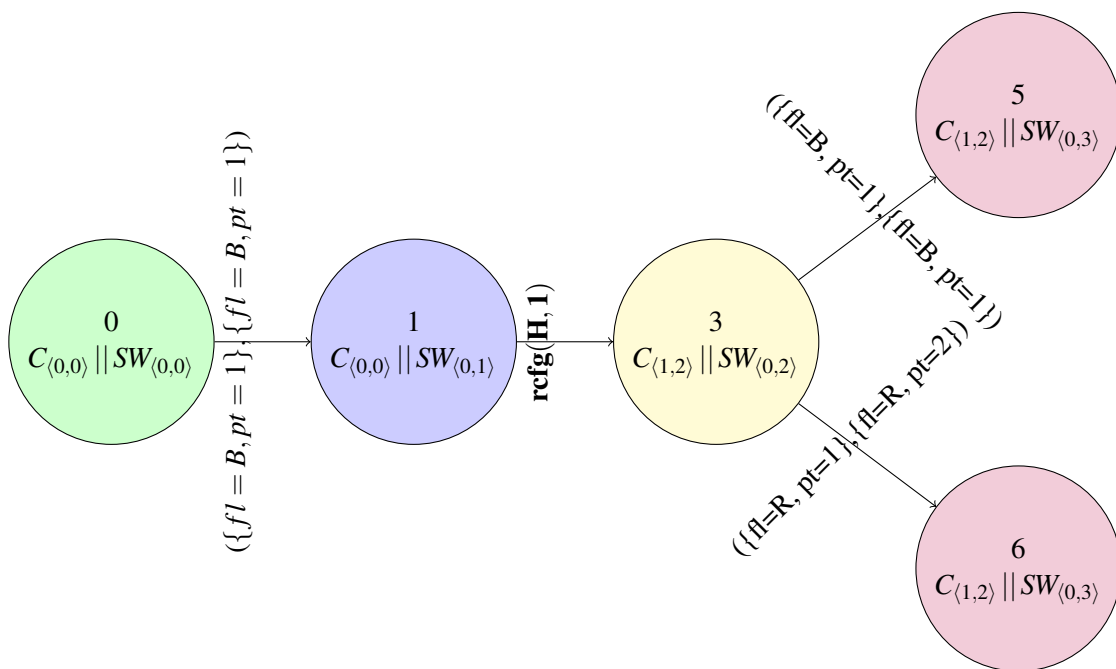
eq getRecPol(SW) =
  "(flag = regular) . (pt = 1) . (pt <- 2)" ; SW o+
  "(flag = blocking) . (pt = 1) . 1" ;( (Help ! "one") ; SW ) o+
  (Up ? "one") ; SWP .
eq SWP = zero ; bot .
eq getRecPol(C) = (Help ? "one") ; ( (Up ! "one") ; C ) .

eq Init = C || SW .
endfm

```

**Listing 1:** Maude encoding of the SDN in (2)

**Tracer Output: Example.** Figure 8 showcases the output races as identified by Tracer, in a graphical format. (We use  $fl, B$  and  $R$  as shorthand for  $flag, blocking$  and  $regular$ , respectively.) The sequence of nodes  $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$  corresponds to the symbolic execution  $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow n_4$  in Figure 7, encoding a race. The sequence of nodes  $0 \rightarrow 1 \rightarrow 3 \rightarrow 6$  in Figure 8 corresponds to the symbolic execution  $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow n_6$  Figure 7, encoding a race as well. Furthermore, the labels along these executions are minimal explanations of how the races can be enabled. Note how the corresponding clocks in Figure 8 match their counterparts in Figure 7.



**Figure 8:** Races in  $SW \parallel C$  up to depth 3

```

1 RACE SHORT TRACES
2 Trace 0:
3 "(flag = blocking) . (pt = 1) "; rcfg('Help', 'one');
4 "(flag = blocking) . (pt = 1)"
5
6 Trace 1:
7 "(flag = blocking) . (pt = 1) . 1"; rcfg('Help', 'one');
8 "(flag = regular) . (pt = 1)"
9
10
11
12 RACE LONG TRACES
13 Trace 0:
14 {C[0, 0] || SW[0, 0]} nid:0;
15 [SW] "(flag = blocking) . (pt = 1) " {C[0, 0] || SW[0, 1]} nid:1;
16 [SW -> C] rcfg('Help', 'one') {C[1, 2] || SW[0, 2]} nid:3;
17 [SW] "(flag = blocking) . (pt = 1)" {C[1, 2] || SW[0, 3]} nid:5;
18
19
20 Trace 1:
21 {C[0, 0] || SW[0, 0]} nid:0;
22 [SW] "(flag = blocking) . (pt = 1) " {C[0, 0] || SW[0, 1]} nid:1;
23 [SW -> C] rcfg('Help', 'one') {C[1, 2] || SW[0, 2]} nid:3;
24 [SW] "(flag = regular) . (pt = 1) " {C[1, 2] || SW[0, 3]} nid:6;

```

**Figure 9:** Race traces of SW || C with unfold 3

Tracer outputs the witnesses of data races in a textual format as well, as illustrated in Figure 9. These traces can be in short form, encoding the input packets and/or reconfiguration steps within symbolic executions without vector clocks. Traces in long form show the action performer (switch SW, controller C or a handshake between the switch and the controller SW → C), the vector clocks ([0, 0], ...), and the corresponding node ID in the graph as well.

## 6 Conclusions

In this paper, we introduced Tracer [1], a tool for detecting and explaining data races in SDNs. These systems exhibit concurrent behavior due to the interaction between data plane processing, and dynamic reconfigurations between the data and control planes. Tracer focuses on pin-pointing data races in SDN models encoded within the DyNetKAT [5] framework. In addition, Tracer provides explanations of how these data races can be enabled by identifying sequences of packets which, whenever fed to the SDN under analysis, lead to concurrency between the data and control planes. The tool is built on top of the DyNetKAT axiomatisation implemented in Maude. In the future, we plan to analyze Tracer's performance on benchmarks with larger SDN models. Additionally, we aim to implement a parallelized version of Tracer to improve its efficiency.

## References

- [1] (2024): *Tracer*. [https://github.com/EZUTwente/DyNetiKAT\\_with\\_race\\_tracing/blob/master/README.md](https://github.com/EZUTwente/DyNetiKAT_with_race_tracing/blob/master/README.md).
- [2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger & David Walker (2014): *NetKAT: semantic foundations for networks*. In Suresh Jagannathan & Peter Sewell, editors: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, ACM, pp. 113–126, doi:10.1145/2535838.2535862. Available at <https://doi.org/10.1145/2535838.2535862>.
- [3] Jos C. M. Baeten & W. P. Weijland (1990): *Process algebra*. *Cambridge tracts in theoretical computer science* 18, Cambridge University Press.
- [4] Georgiana Caltais, Nate Foster & Hossein Hojjat (2024): *Symbolic Race Identification in DyNetKAT*. <https://drive.google.com/file/d/1UdpsgqhoRTzNTKKi6RJeLCZTLuobVwst/view>.
- [5] Georgiana Caltais, Hossein Hojjat, Mohammad Reza Mousavi & Hünkar Can Tunç (2022): *DyNetKAT: An Algebra of Dynamic Networks*. In Patricia Bouyer & Lutz Schröder, editors: *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Lecture Notes in Computer Science* 13242, Springer, pp. 184–204, doi:10.1007/978-3-030-99253-8\_10. Available at [https://doi.org/10.1007/978-3-030-99253-8\\_10](https://doi.org/10.1007/978-3-030-99253-8_10).
- [6] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever & Martin T. Vechev (2016): *SDNRacer: concurrency analysis for software-defined networks*. In Chandra Krintz & Emery D. Berger, editors: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, ACM, pp. 402–415, doi:10.1145/2908080.2908124. Available at <https://doi.org/10.1145/2908080.2908124>.
- [7] James Hamilton (2009): *Networking: The last bastion of mainframe computing*. Available at <https://perspectives.mvdirona.com/2009/12/networking-the-last-bastion-of-mainframe-computing/>.
- [8] Dexter Kozen (1997): *Kleene Algebra with Tests*. *ACM Trans. Program. Lang. Syst.* 19(3), pp. 427–443, doi:10.1145/256167.256195. Available at <https://doi.org/10.1145/256167.256195>.
- [9] Leslie Lamport (2019): *Time, clocks, and the ordering of events in a distributed system*. In Dahlia Malkhi, editor: *Concurrency: the Works of Leslie Lamport*, ACM, pp. 179–196, doi:10.1145/3335772.3335934. Available at <https://doi.org/10.1145/3335772.3335934>.
- [10] Xiaoye Steven Sun, Apoorv Agarwal & T. S. Eugene Ng (2015): *Controlling Race Conditions in OpenFlow to Accelerate Application Verification and Packet Forwarding*. *IEEE Transactions on Network and Service Management* 12(2), pp. 263–277, doi:10.1109/TNSM.2015.2419975.
- [11] Andrew S. Tanenbaum & Maarten Van Steen (2007): *Distributed Systems: Principles and Paradigms*, 2nd edition. Pearson Prentice Hall, Upper Saddle River, NJ. Available at [https://vowi.fsinf.at/images/b/bc/TU\\_Wien-Verteilte\\_Systeme\\_VO\\_%28GÃ¼schka%29\\_-\\_Tannenbaum-distributed\\_systems\\_principles\\_and\\_paradigms\\_2nd\\_edition.pdf](https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_%28GÃ¼schka%29_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf).
- [12] Evgenii M. Vinarskii, Jorge López, Natalia Kushik, Nina Yevtushenko & Djamel Zeglache (2019): *A Model Checking Based Approach for Detecting SDN Races*. In Christophe Gaston, Nikolai Kosmatov & Pascale Le Gall, editors: *Testing Software and Systems - 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15-17, 2019, Proceedings, Lecture Notes in Computer Science* 11812, Springer, pp. 194–211, doi:10.1007/978-3-030-31280-0\_12. Available at [https://doi.org/10.1007/978-3-030-31280-0\\_12](https://doi.org/10.1007/978-3-030-31280-0_12).
- [13] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang & Guofei Gu (2017): *Attacking the Brain: Races in the SDN Control Plane*. In Engin Kirda & Thomas Ristenpart, editors: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, USENIX Association, pp. 451–468. Available at <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-lei>.