Automated detection of race conditions in DyNetKAT

ERVIN ZVIRBULIS, University of Twente, The Netherlands

Traditional network architectures have relied on special-purpose hardware since the 1970s. Such an approach limits network adaptability and increases set-up time. Software-Defined Networking (SDN) offers a solution by centralizing control over the network switches. One of the tools for modeling networks is DyNetKAT, an extension of NetKAT. Both are network modeling languages based on the Kleene algebra with tests, a mathematical framework used to model and analyze the behavior of systems encoded as regular expressions with tests.

DyNetKAT is catered specifically to model SDNs. However, it has limited capabilities for tracing data races - unexpected network behavior caused by communication delays between the central control point and and the forwarding plane (the switches). The aim of the research is to create a tool to solve this issue. By using vector clocks, the paper seeks to design and develop a solution for automated data race detection.

This addition will expand the capabilities of DyNetKAT, improving the stability of the networks modeled and validated with the Tracer developed in this paper, by pointing out potential issues. Furthermore, the results have the potential to facilitate a deeper discussion and examination of the root causes of data races in SDNs.

Additional Key Words and Phrases: SDN (software defined networking), vector clocks, (Dy)NetKAT, data races, control plane, data plane, network modelling, traceability, dynamic reconfiguration, network behavior

1 INTRODUCTION

1.1 Common network setup

Traditional network devices have been called “the last bastion of mainframe computing” [10]. 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 plane[1] and control plane[2]. In a traditional network, each switch autonomously manages its interpretation of the control plane (Figure 1).

This architectural rigidity increases complexity in network maintainability due to the necessity of configuring each switch individually. As well as making it effectively impossible to reason precisely about network behaviors, because of the large amount of individual asynchronous components.

1.2 Software defined network

In response, the concept of software-defined networking (SDN) has emerged. With the main differential being the separation of the data and control planes and consolidation of the management over the control plane in a centralized location (Figure 2). SDN architecture comprises a central controller and programmable switches that communicate via standardized protocols. The former responds to network events such as new connections from hosts, topology changes, and shifts in traffic load by re-programming the switches accordingly [1]. Such an approach enhances network controllability and adaptability in real-time scenarios.

[1]A distinct functional layer in networking responsible for the forwarding of data packets between network devices.

[2]A distinct functional layer in networking responsible for network control including policy enforcing and routing configuration.

TScIT 41, July 5, 2024, Enschede, The Netherlands

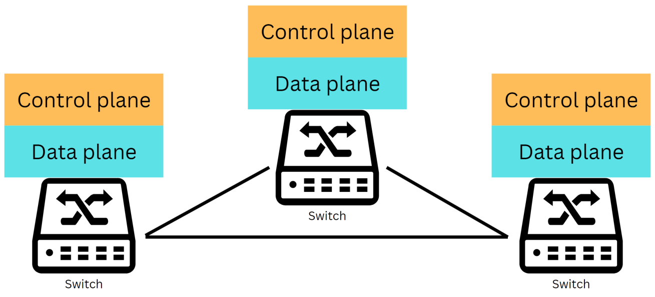

Fig. 1. Traditional network setup

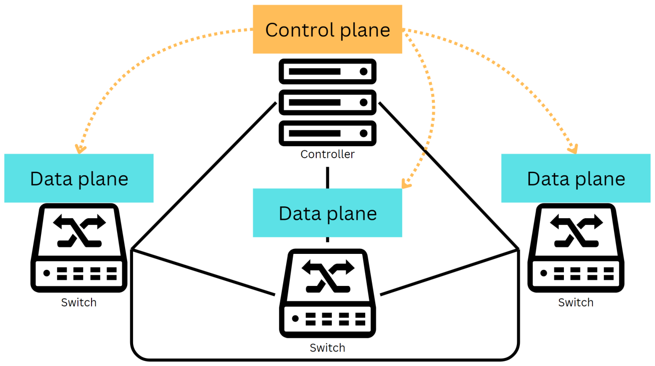

Fig. 2. Software-defined network setup

1.3 Modeling framework

The research necessitates a model to simulate a network and detect possible data races in a repeatable, controlled environment.

1.3.1 NetKAT. A sound foundation is NetKAT, a domain-specific language for specifying, programming, and reasoning about network forwarding behavior based on Kleene algebra[3] with tests[4], it provides a foundational structure and consistent reasoning principles [1]. However, NetKAT does not support dynamic reconfiguration of the network, limiting each component to a constant set of

[3]A mathematical structure used to model and analyze the behavior of regular expressions and formal languages, particularly focusing on operations like union, concatenation, and Kleene star (iteration) [2].

[4]An extension of Kleene algebra that includes Boolean tests, allowing for the modeling and analysis of both program control flow and data flow within a unified algebraic framework [11].

Table 1. DyNetKAT operations

Operator	Explanation
f = n	**Filter** packets with filed ***f*** that match value **n**
f←n	**Modify** packets field ***f*** with value **n**
p . q	**Apply** operations ***p*** and ***q*** to the packet
;	**Sequential composition** denotes the end of packet operations
Ch ! m	(A)synchronously **send** message ***m*** on a ***Ch*** channel can happen only in conjunction with ***?*** from a different component
Ch ? m	(A)synchronously **receive** message ***m*** on a ***Ch*** channel can happen only in conjunction with ***!*** from a different component
⊕	**Non-deterministic choice** between two sequences actions
⊥	No (**null**) action
1	**Accept** / true
0	**Reject** / false

rules. This makes it practically impossible to model concurrent SDN behaviors.

1.3.2 DyNetKAT. To remedy these shortcomings, DyNetKAT [3] was introduced as an extension. Unlike NetKAT, DyNetKAT was built with dynamic SDNs in mind. It enables the ability to model communication between data and control planes, allowing it to properly represent an SDN.

DyNetKAT is powerful and relatively simple to use due to its high level of abstraction and Kleene completeness[5]. Furthermore, as the focus of this study revolves around enhancing the capabilities of DyNetKAT, it inherently dictates the utilization of DyNetKAT as the primary framework for investigation and improvement.

1.4 Running model

Figure 3 illustrates an SDN setup with one controller, one switch, and two hosts. The size and simplicity of the model allow for an easier presentation in this paper while still capturing the potential for data races.

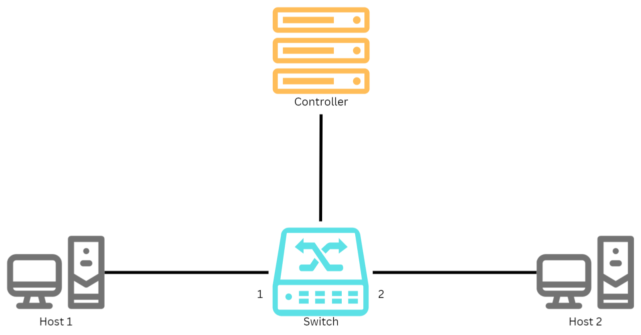

Fig. 3. Simple SDN setup

Equation (1) defines two DyNetKAT components (*Switch* and *Controller*) that are executing in parallel:

$$SW \quad || \quad C \tag{1}$$

[5]A property of a language indicating that it can express all regular behaviors using the operations of union, concatenation, and Kleene star, fully capturing the patterns and sequences described by regular expressions and finite automata.

Next, the rules of operation are defined per component (refer to Table 1 for operator explanation). It should be noted that in this paper send and receive are used synchronously (handshake communication). Whenever such communication happens, the event is marked as '*rcfg*'.

Equation (2) defines the *Switch*:

$$\begin{aligned} SW := & (flag = \text{regular}) \cdot (pt = 1) \cdot (pt \leftarrow 2); \quad SW \oplus \\ & (flag = \text{blocking}) \cdot (pt = 1) \cdot 1; \quad (\,(Help \,!\, 1);\ SW) \oplus \\ & (Up \,?\, 1); \quad SW' \end{aligned} \tag{2}$$

It has a simple configuration. It recognizes two types of packets, blocking and regular (stored in the flag field of the packet). Both must come from Host 1 and only *regular* packets will be forwarded to Host 2. If a *blocking* packet is encountered, the *Switch* asks the *Controller* for help by sending a message on channel 'Help'.

The rules define three non-deterministic choices or options.

- **Choice 1**
  1. Manipulate and forward the message packet[6]:
     1. $(flag = \text{regular})$ - check if a packet has field *flag* equal to value 'regular'.
     2. $(pt = 1)$ - check if the packet is at port 1.
     3. $(pt \leftarrow 2)$ - forward the packet to port 2;
  2. Behave like itself (choose from initial choices);
- **Choice 2**
  1. Manipulate and forward the message packet:
     1. $(flag = \text{blocking})$ - heck if packet has field *flag* equal to 'blocking'.
     2. $(pt = 1)$ - same as in Choice 1.
     3. 1 - accept the packet, but not forward it;
  2. Send message "1" on channel '*Help*';
  3. Behave like itself;
- **Choice 3**
  1. Receive message "1" on channel '*Up*';
  2. Dynamically reconfigure to rules of *Switch*‘;

Note that steps within a choice must execute consecutively, while choices themselves can happen arbitrarily. For example, imagine that *Switch* is in choice 1. It manipulates the incoming packet, forwards it, and moves on to step 2. At this point, the *Switch* can behave in accordance with choice 1, 2, or 3, depending on the packets in the network.

If the *Switch* gets the message on channel 'Up', it should reconfigure to the alternative mode of operation:

$$SW' := 0; \quad \perp \tag{3}$$

Equation (3) represents reconfiguration of the *Switch*, for example policy or routing table updates. In this particular case this definition states that the *Switch* must do nothing.

Last but not least, controller can repeatedly receive "1" on 'Help' channel, then send the message "1" on channel 'Up' (respond to the message on 'Help'):

$$C := (Help \,?\, 1); \quad (\,(Up \,!\, 1);\ C) \tag{4}$$

[6]If all packet field comparisons and reassignments are successful, the packet is forwarded, otherwise it is ignored.

Note that both the *Controller* (Equation (4)) and the *Switch* (Equation (2)) are defined recursively. Such definitions allow to model infinite continuous operations. Notice that the definition of the *Controller* is truly endless. In contrast the *Switch* will end its execution if reconfigured to *Switch'* since the rules do not specify any action after $\perp$.

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

2 RELATED WORK

There is a multitude of different modeling languages for SDNs [5, 7, 9, 13–15, 17, 18], however only NetKAT (and its extensions) are Kleene complete, meaning that it is possible to fully reason about the system behavior using such language. This capability allows network administrators to define, analyze, and enforce policies effectively, thereby ensuring reliable and efficient network operation.

The main existing alternative for this article is the SDNRacer [4]. It employs happens-before models[7] for reasoning about event ordering, whereas DyNetKAT utilizes Kleene algebra for network modeling and analysis. The big difference between the two approaches is that while the SDNRacer tests the network model by providing inputs based on real-world logs, the DyNetKAT Tracer determines the network inputs based on its definition, which theoretically simplifies and accelerates the validation process.

3 MOTIVATION

Although the DyNetKAT framework is catered to dynamic SDNs, it has its limitations. This paper focuses on two such hindrances - the detection and the traceability of data races within a network. The research aims to create a tool to address these drawbacks by creating an extension of DyNetKAT. The tool will improve the existing framework with a validation process against data races, increasing the stability of the networks validated with DyNetKAT.

4 CONTRIBUTIONS

- Easy installation of DyNetKAT using single script (install.sh from research repository).
- Design of the algorithm for data race detection (Section 7).
- Implementation of the designed algorithm (TracerTool in research repository).

5 DATA RACES

5.1 What are data races

In networking, a race condition occurs when the timing and order of events, such as packet arrivals or thread executions, lead to unpredictable and inconsistent system behavior.

This research focuses on data races, specifically between the data and control planes. That is when a switch or a number of them use an outdated policy and/or forwarding table due to communication delays between the controller and the switch.

5.2 Representation in the running model

If we take a closer look at the *Switch*, we can see that the second non-deterministic choice includes communication on the channel '*Help*'. The *Controller* is set up to respond on another channel, which when received by the *Switch* changes its behavior to *Switch'*. This effectively makes the act of sending on the channel '*Up*' take the role of the rule update to the *Switch*.

The sequence:

$$\begin{aligned} &\textit{Switch} \text{ requests an update} \rightarrow \textit{Controller} \text{ receives the request} \rightarrow \\ &\qquad \textit{Controller} \text{ responds with the update} \rightarrow \textit{Switch} \text{ updates} \end{aligned}$$

can be defined as the update communication. And if completed atomically[8] both components will stay synchronized.

By the definition of the *Switch* (Equation (2)) it is possible to receive a response (an update) or receive a data packet. The latter simulates a communication delay between the *Controller* and the *Switch*, allowing the *Switch* to follow outdated rules. This is a data race - the *Switch* performs an action during the update communication.

5.3 How to detect data races

SDN is a paradigm that falls under the definition of a distributed system[9]. In this case, the components are controllers and switches, and the whole network represents a complete system.

In distributed systems like SDN, a data race means that the components of a system are out of sync. So, race detection necessitates a mechanism to determine whether components are synchronized. One such mechanism for distributed systems is the use of vector clocks.

6 VECTOR CLOCKS

6.1 Basics

It is a mechanism used in distributed systems to order events [6, 12].

As the name implies, it consists of a vector with a size equal to the number of components in the system, where each vector entry represents a step counter (clock) for each distinct component. Each component in the system has its own copy of a vector clock (Figure 4).

6.2 Inner workings

As shown in Figure 4, step 0 involves initiating all clocks with zeros. When a component has an internal event, 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 (step 1). Both steps 2 and 3 are the same as just described but to another component.

Once a component sends a message, it first increments its clock, and then sends it along with the message, creating a timestamped

[7] A conceptual framework used to reason about the temporal ordering of events in a distributed system.

[8] An indivisible and irreducible sequence of operations that must be executed completely or not executed at all.

[9] A distributed system is a collection of independent computers that appears to its users as a single coherent system [16].

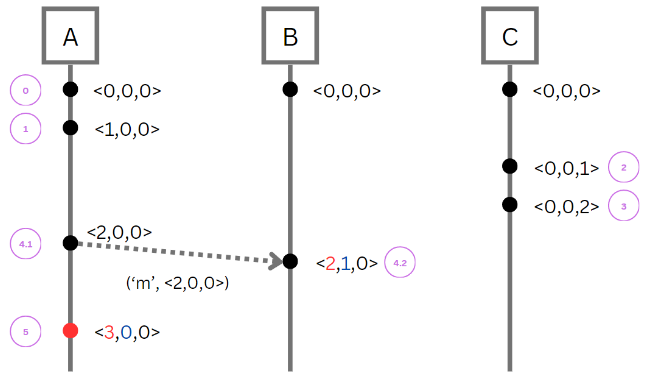

Fig. 4. Vector clocks in a system with the size 3 (three components)

message (step 4.1). Upon its arrival, the receiver updates each entry in its own clock with the corresponding entry from the message timestamp if the stamp has a greater value than the local copy, then it increments its clock entry by one (step 4.2). Note that step 4 is split in two, this is because in DyNetKAT communication is synchronous, meaning send and receive happen in one time frame[10].

6.3 Race detection using vector clocks

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.

Lets take two vector clocks V_i and V_j, array of ids IDs, and two components ids x and y. The clocks are comparable if:

$$V_i[x] \leq V_j[x] \qquad \forall x \in IDs$$

or

$$V_i[x] \geq V_j[x] \qquad \forall x \in IDs$$

Such pair of vector clocks indicate that the components are synchronized. In the case of:

$$\exists\,(V_i[x] < V_j[x] \quad \wedge \quad V_i[y] > V_j[y]) \quad \text{, where } x, y \in IDs \wedge x \neq y$$

two clocks suggest that the components are operating in parallel and are no longer synchronized, implying a data race. Figure 5 visualizes comparable in incomparable vector clocks. While Figure 4, step 5, showcases one possible scenario of vector clocks becoming incomparable. Notice that step 5 is not the only race condition in the example, however, races without communication between the components are generally fine since the switches can transmit data packets without synchronization.

7 TRACING ALGORITHM

This algorithm is the core of the tool. It is designed to ingest a DyNetKAT model, then iterate over all possible execution paths until either a recursion depth limit is reached or a data race has occurred. The output consists of execution traces that caused the violation, which themselves are sequences of network events.

[10]This is not true for every distributed system.

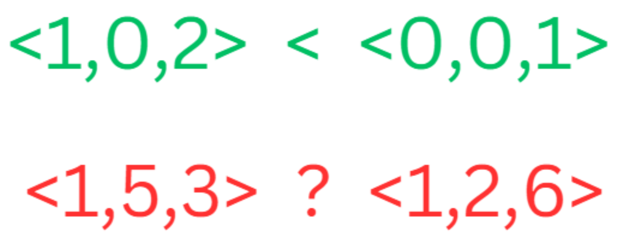

Fig. 5. Comparable (green) and incomparable (red) clocks

7.1 Critical DyNetKAT axioms

Two fundamental DyNetKAT axioms for the algorithm are $pi\{m\}(Q)$ and *reduce* Q. The *pi* operator takes n as the unfold limit and Q as the DyNetKAT expression. It unfolds recursive definitions n times, transforming infinite definition into a finite subset of actions, where n is the number of actions separated by sequential composition (;). It is important to recognize that packet operators (=, ⟵ and ·) together until ';' are considered as one action.

$$\begin{aligned}SW := &(flag = \text{regular}) \cdot (pt = 1) \cdot (pt \leftarrow 2); (\\ &\qquad (flag = \text{regular}) \cdot (pt = 1) \cdot (pt \leftarrow 2); \quad \bot \oplus \\ &\qquad (flag = \text{blocking}) \cdot (pt = 1) \cdot 1; \quad \bot \oplus \\ &\qquad (Up\ ?\ \text{"}one\text{"}); \quad \bot) \\ &\quad \oplus (flag = \text{blocking}) \cdot (pt = 1) \cdot 1; \quad (\ (Help\ !\ \text{"}one\text{"});\ \bot) \\ &\quad \oplus (Up\ ?\ \text{"}one\text{"}); \quad \bot\end{aligned} \tag{5}$$

Equation 5 show application of $pi\{2\}(SW)$. Note that each non-deterministic choice consists of two actions and the end statement separated by ';'. The third choice is the exception, because the rules from Equation 3 simply state to stop execution.

Since the implementation of DyNetKAT is written in Maude[11] *reduce* command is responsible for applying DyNetKAT axioms (rules). It transforms a complex definition into a single chain expression, representing possible execution routes.

The *pi*, just like any other operator must be used in conjunction with *reduce* to get the desired output.

$$reduce\ pi\{2\}(SW)$$

It is worth mentioning that DyNetKAT terms can be separated into three groups:

- $(flag = \text{regular}) \cdot (pt = 1) \cdot (pt \leftarrow 2);$ - max-text-assignments (block of packet operations, namely =, ⟵ and ·).
- $(Help\ !\ \text{"}one\text{"})$ - send
- $(Help\ ?\ \text{"}one\text{"})$ - receive

7.2 Variables

- $N_1, N_2, ..., N_k$ - components of the network (controllers and switches) with id 'k'.
- $C_k :=< 0, 0, ..., 0 >$ - vector clock of the component 'k'.

[11]Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications.

```
load ../../src/maude/dnk.maude

fmod MODEL is
   protecting DNA .
   protecting PROPERTY-CHECKING .

   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
```

Fig. 6. Model in .maude

- $P := (N_1\ C_1) \mid\mid (N_2\ C_2) \mid\mid ... \mid\mid (N_k\ C_k)$ - the network configuration (program) where 'k' components with vector clocks are executing in parallel.
- $pmN_k := reduce\ pi\{m\}(N_k)$ - short notation for reducing component N_k unfolded with limit m.
- mts - max-test-assignments term
- $\mathbf{D} := \{mts \text{ or } send \text{ or } receive \text{ or } \bot\} ; \mathbf{D}$ - recursive format of the expression after unfolding.

7.3 Steps

The algorithm starts with the initial program P. Then applies *reduce pi* to each component and creates a starting node with an unfolded program: $(pmN_1\ C_1) \mid\mid (pmN_2\ C_2) \mid\mid ... \mid\mid (pmN_k\ C_k)$.

For each pair of components $(N_i\ C_i)$ and $(N_j\ C_j)$ $\forall i, j \in \{1, 2, ..., k\}$, check if C_i and C_j are comparable. If so, continue, otherwise, save the execution path as the trace and exit the node[12]. If the first action of the expression is in form 'mts ; D':

- record mts as part of the execution;
- update clock C_i to C_i' (increment the element at index i in C_i);
- replace pmN_i with D_i (the same action sequence as pmN_i, excluding the first action), resulting in: $P' := (pmN_1\ C_1) \mid\mid (pmN_2\ C_2) \mid\mid ... \mid\mid (D_i\ C_i) \mid\mid ... \mid\mid (pmN_k\ C_k)$
- create a new child node with P' as the updated program.

For all pairs of elements $(pmN_i\ C_i)$ and $(pmN_j\ C_j)$[13], if $(X\ !\ msg;\ D_i)$ is part of the pmN_i and $(X\ ?\ msg;\ D_j)$ is part of the pmN_j:

- record $rcfg(X,\ msg)$ as part of the execution;
- the receiver (N_j) updates its clock (increment the element at index j and update the rest of the entries in C_j if the corresponding ones in C_i are greater);
- update the program to: $P' := (pmN_1\ C_1) \mid\mid (pmN_2\ C_2) \mid\mid ... \mid\mid (D_i\ C_i) \mid\mid ... \mid\mid (D_j\ C_j) \mid\mid ... \mid\mid (pmN_k\ C_k)$[14]
- create a new child node with P' as the updated program.

Otherwise if pmN_i is $\bot$, skip the element.

The steps are repeated for each created child recursively from clock comparison (start of previous paragraph), until no more children can be created.

7.4 Input format

The encoding shown before is purely theoretical, actual DyNetKAT implementation[15] requires the model to be in .maude format, which means that the Maude syntax must be used. Figure 6 depicts the same running model in Maude implementation. The 'DNA', 'PROPERTY-CHECKING', 'Recursive', and 'Channel' are modules and types required by DyNetiKAT. *getRecPol* is a special operator, its purpose is to define recursive definitions. Be aware that max-test-assignments and all packet related operations are in a string format.

The model along with the recursion limit are required as the input.

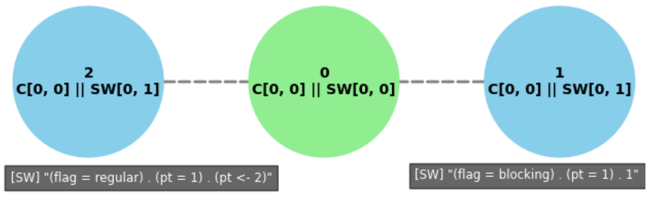
2
C[0, 0] || SW[0, 1]
0
C[0, 0] || SW[0, 0]
1
C[0, 0] || SW[0, 1]
[SW] "(flag = regular) . (pt = 1) . (pt <- 2)"
[SW] "(flag = blocking) . (pt = 1) . 1"

Fig. 7. Execution tree of program $SW \mid\mid C$ with unfold 1 (the starting node is marked in green)

[12]This will not happen on the first pass, but is an important recursion end condition.
[13]If send or receive do not have a compatible pair (same channel and same message), they are skipped (ignored).

[14]Notice that C_i and C_j are symmetric with respect to $\mid\mid$, meaning $C_i \mid\mid C_j$ is the same as $C_j \mid\mid C_i$.
[15]DyNetiKAT available at https://github.com/hcantunc/DyNetiKAT

8 VALIDATION

To validate the correctness of the tool, we can check if the produced execution trees match expectations. Running the tracer with unfold 1 yields an execution tree with two child nodes representing two possible options: getting a packet with flag regular or blocking (Figure 7).

This matches the definitions of the *Switch* (Equation 2) and the *Controller* (Equation 4). Out of the three possible *Switch* choices, the third is the only one that necessitates a preceding action from another component (*Controller*). While the *Controller* can not perform the sending action, without receiving first. Both actions were taken by the *Switch*, so in both nodes only its clock was updated.

Up until unfold of 3, no data races occur, so no race traces are produced. Figure 8, showcases the first race tree. As expected, the

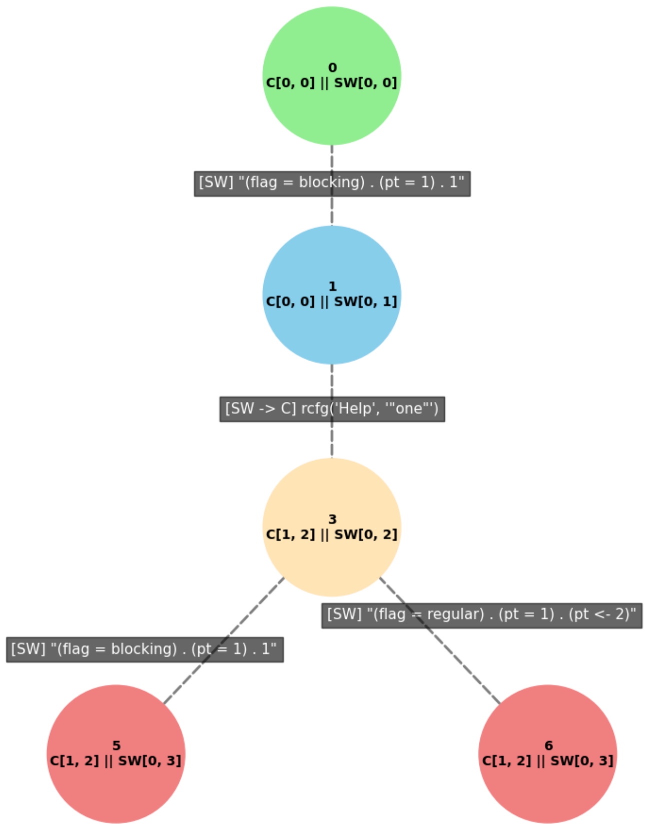
0
C[0, 0] || SW[0, 0]
[SW] "(flag = blocking) . (pt = 1) . 1"
1
C[0, 0] || SW[0, 1]
[SW -> C] rcfg('Help', '"one"')
3
C[1, 2] || SW[0, 2]
[SW] "(flag = blocking) . (pt = 1) . 1"
[SW] "(flag = regular) . (pt = 1) . (pt <- 2)"
5
C[1, 2] || SW[0, 3]
6
C[1, 2] || SW[0, 3]

Fig. 8. Race execution tree of $SW \mid\mid C$ with unfold 3

race condition occurred after reconfiguration (Node 3), or $(Help\ !\ 1)$ from the *Switch* and $(Help\ ?\ 1)$ from the *Controller*. When the former performs a packet-related action, instead of completing the handshake. Keep in mind that the race tree is a sub-tree of the full execution path. You can find a full graph with unfold 3 and more in the repository[16], specifically in the directory 'OUTPUTS_FOR_PAPER'[17].

[16]https://github.com/EZUTwente/DyNetiKAT_with_race_tracing

[17]https://github.com/EZUTwente/DyNetiKAT_with_race_tracing/tree/master/OUTPUTS_FOR_PAPER

The resulting traces can be seen in the Figure 9 in the short form as a single expression, and in the long form showing the performer, the action, the vector clocks, and the corresponding node id in the graph. All traces include only one $rcfg$ step, meaning the update was not completed and the data races between the planes has occurred. Both traces describe the expected, result, which validates the implementation.

9 DISCUSSION

9.1 Usage and installation

The *DyNetiKAT_with_tracer* repository contains the instructions on how to install and run the tracer in the README.md file in the root directory. The only system requirement is the Linux operating system, specifically Ubuntu 20.04[18] with python (version > 3.10.12). The tool also uses Maude 3.1, however it is included in the installation of the 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 (Table 2):

Table 2. DyNetKAT operations

Parameter	Value	Explanation
-c	-	output text with color
-t	-	show tracing steps
-u	int	unfold depth
-g	'race' or 'full'	choose which tree to generate and what traces to print
-f	string	set a name for text output file (copy of console output)

Note that the parameters with values shoud be inputed without the space between the parameter and the value (example '-grace' to produce only race graphs and traces)

9.2 Implementation

The resulting extension called Tracer (available at `github.com/EZUTwente/DyNetiKAT_with_race_tracing`) met the assigned goal - implement race tracing tool for DyNetKAT (based on the theory in [8]). It is, however, a sophisticated prototype, rather than the final product. The main limitation is the speed, which is caused by the sub-optimal efficiency.

It was not important in this step of the broader DyNetKAT and SDN research. However, it is extremely important for the power and the performance of the algorithm. The two slowest parts of the tool are the tree creation/traversal and the clock comparison. The former can be improved with parallelization, since the branches of each node and independent between themselves. This will allow for multiple CPU threads to explore several execution paths simultaneously, dramatically increasing the speed. The latter can be addressed with a better comparison algorithm, as well as with parallelization. The current comparison mechanism uses two-dimensional iteration, checking every pair, resulting in the exponential complexity of $O(n^2)$, where n is the number of components. As the comparison is done pairwise, it can also benefit from parallel execution. One

[18]Other Linux distributions might work, however, the development and testing were done on the specified version of Ubuntu.

```
RACE SHORT TRACES
Trace 0:
"(flag = blocking) . (pt = 1) . 1"; rcfg('Help', '"one"'); "(flag = blocking) . (pt = 1) . 1";

Trace 1:
"(flag = blocking) . (pt = 1) . 1"; rcfg('Help', '"one"'); "(flag = regular) . (pt = 1) . (pt <- 2)";



RACE LONG TRACES
Trace 0:
{C[0, 0] || SW[0, 0]} nid:0;
[SW] "(flag = blocking) . (pt = 1) . 1" {C[0, 0] || SW[0, 1]} nid:1;
[SW -> C] rcfg('Help', '"one"') {C[1, 2] || SW[0, 2]} nid:3;
[SW] "(flag = blocking) . (pt = 1) . 1" {C[1, 2] || SW[0, 3]} nid:5;


Trace 1:
{C[0, 0] || SW[0, 0]} nid:0;
[SW] "(flag = blocking) . (pt = 1) . 1" {C[0, 0] || SW[0, 1]} nid:1;
[SW -> C] rcfg('Help', '"one"') {C[1, 2] || SW[0, 2]} nid:3;
[SW] "(flag = regular) . (pt = 1) . (pt <- 2)" {C[1, 2] || SW[0, 3]} nid:6;
```

Fig. 9. Race traces of $SW \mid\mid C$ with unfold 3 (each action is colored differently for visibility)

should acknowledge that the parallelization is not the silver bullet for efficiency drawbacks. A CPU can have only so many cores and threads, so the use of more efficient algorithms and/or data structures is most likely unavoidable.

10 CONCLUSIONS

The research successfully achieved the assigned goals, namely to design and implement the tracing algorithm as an extension for the DyNetKAT framework. The prototype is not perfect but suffices to prove the concept and provides a solid foundation for the future improvements mentioned in the implementation section. If refined, it can help improve SDN stability validated with the Tracer tool, by indicating potential problems before deployment. This will allow for future race research, and provide potential real-world usage.

11 USE OF AI TOOLS

This paper made use of AI tools such as Grammarly for the grammar checking. As well as OpenAI ChatGPT and Microsoft Copilot as complementary search engines and dictionaries.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude towards several people for their invaluable contribution to this research. First and foremost, I am deeply thankful to my supervisors Georgiana Caltais and Mahboobeh Zangiabady for their expertise, introduction into SDNs and network data races, support in the design of the core algorithm, time management, and the overall mentorship and guidance during the research.

I would also like to thank my track chair Suzan Bayhan for the insightful feedback and the organizational aspects.

REFERENCES

[1] 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. https://ecommons.cornell.edu/server/api/core/bitstreams/a15a86f5-2d0e-4361-a384-ba136bfd1b36/content

[2] Alasdair Armstrong, Georg Struth, and Tjark Weber. 2013. Kleene algebra. *Archive of Formal Proofs* 324 (2013). https://dl.acm.org/doi/pdf/10.1145/256167.256195

[3] Georgiana Caltais, Hossein Hojjat, Mohammad Reza Mousavi, and Hünkar Can Tunç. 2022. DyNetKAT: An Algebra of Dynamic Networks. In *Foundations of Software Science and Computation Structures*, Patricia Bouyer and Lutz Schröder (Eds.). Springer International Publishing, Cham, 184–204. https://link.springer.com/chapter/10.1007/978-3-030-99253-8_10#Abs1

[4] 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*. 402–415. https://pavol-bielik.github.io/data/papers/pldi16-sdn.pdf

[5] Andrew D Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2013. Participatory networking: An API for application control of SDNs. *ACM SIGCOMM computer communication review* 43, 4 (2013), 327–338. https://dl.acm.org/doi/pdf/10.1145/2486001.2486003

[6] Colin J Fidge. 1987. Timestamps in message-passing systems that preserve the partial ordering. (1987). https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=e706b8ae2952740cb95c0182c4c44b0d11cc54c1

[7] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. *ACM Sigplan Notices* 46, 9 (2011), 279–291. https://oar.princeton.edu/bitstream/88435/pr17r8s/1/FreneticNetworkProgrammingLang.pdf

[8] Hossein Hojjat Georgiana Caltais, Nate Foster. 2024. Symbolic Race Identification in DyNetKAT. (2024). https://drive.google.com/file/d/1UdpsgqhoRTzNTKKi6RJeLCZTLuobVwst/view

[9] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-verified network controllers. *Acm Sigplan Notices* 48, 6 (2013), 483–494. https://www.cs.cornell.edu/~jnfoster/papers/frenetic-verified-controllers.pdf

[10] James Hamilton. 2009. Networking: The last bastion of mainframe computing. (2009). https://perspectives.mvdirona.com/2009/12/networking-the-last-bastion-of-mainframe-computing/

[11] Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (1997), 427–443. https://dl.acm.org/doi/pdf/10.1145/256167.256195

[12] Friedemann Mattern et al. 1988. *Virtual time and global states of distributed systems.* Univ., Department of Computer Science. https://nylas.github.io/paper-reading-

group/papers/Virtual_Time.pdf

[13] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A compiler and run-time system for network programming languages. *Acm sigplan notices* 47, 1 (2012), 217–230. http://frenetic-lang.org/publications/compiler-popl12.pdf

[14] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 1–13. https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final232.pdf

[15] Tim Nelson, Arjun Guha, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2013. A balance of power: Expressive, analyzable controller programming. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. 79–84. https://dl.acm.org/doi/pdf/10.1145/2491185.2491201

[16] Andrew S. Tanenbaum and Maarten Van Steen. 2007. *Distributed Systems: Principles and Paradigms* (2nd ed.). Pearson Prentice Hall, Upper Saddle River, NJ. https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_%28GĂűschka%29_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf

[17] Andreas Voellmy and Paul Hudak. 2011. Nettle: Functional reactive programming of OpenFlow networks. *PADL, Jan* (2011). https://courses.cs.duke.edu/compsci590.4/fall13/838-CloudPapers/Nettle.pdf

[18] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. 2013. Maple: Simplifying SDN programming using algorithmic policies. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 87–98. https://dl.acm.org/doi/pdf/10.1145/2486001.2486030