Causal Analysis of Safety Violations in DyNetKAT

CAN OLMEZOGLU, University of Twente, The Netherlands

Failures in concurrent systems, particularly software defined networking (SDN) systems can be very difficult to detect for the human brain. In order to deal with such failures computationally, we introduce a system to explain and debug safety failures in SDNs expressible in DyNetKAT. The latter is a framework to represent the control and data plane of SDNs, based on a well-researched network programming language, NetKAT (Network Kleene Algebra with Tests). We provide and implement an algorithm to convert DyNetKAT into a Labelled Transition System (LTS), based on the Maude Rewriting Logic and the NetKAT tool. We exploit the counterfactual causal reasoning theory on the generated LTS to understand which transitions led to the safety failure. We show that the safety failures of the running examples can be identified and explained using our prototype tool.

Additional Key Words and Phrases: Software Defined Networks, Causality, Formal Specification

1 Introduction

The main objective of an engineer is to build systems which follow a predefined behaviour. Explaining when a system fails to follow through that behaviour has thereby gained a lot of attention as engineering rose to prominence. In order to understand and debug such failures, Fault Tree Analysis (FTA) [17] and Failure mode and effects analysis (FMEA) [8] have been proposed and widely used.

Starting with Hume's work [13], there has been a lot of research on causal reasoning, which can ultimately be utilized in explaining system failures. The work by David Lewis proposed the alternative worlds idea. Briefly, Lewis proposes in order for a cause to have a causal relationship with an effect, worlds where a sufficiency and necessity condition are satisfied has to exist. To satisfy the former, defined as the sufficiency condition, whenever the cause happens, the effect also has to happen, and to satisfy the latter, defined as the necessity condition, whenever the cause does not happen, the effect should not happen as well [19]. Nevertheless, this has been considered as too simple when faced with the logical complexities that arise with system failures [4]. The definition of causality adopted by computer scientists [20] is the definition given by Halpern and Pearl in their landmark studies [11] [10] .

TScIT 37, July 8, 2022, Enschede, The Netherlands

Amongst others, this definition on top of the sufficiency and necessity conditions previously mentioned, adds minimality requirement and captures aspects like preemption. Minimality requirement assures that only the proper group of causal occurrences are recognized, where the proper group can be defined as a group solely consisting of relevant causal events. Preemption involves a group of events which all have ability to enable to effect, where temporal differences decide which one of them actually does enable the effect [10].

In this paper, we focus on understanding and debugging system failures in software defined networks (SDNs) using causality. Software defined networking has gained a lot of traction as it has simplified network management and offered network programmability by decoupling the control plane from the data plane and adding an application programming plane in between the control plane and the data plane [14]. Following this, programming languages such as OpenFlow [21] and Frenetic [9] have been designed for SDNs. Over the recent years, formal languages for SDNs have become more acclaimed as they allow for reasoning about correctness properties such as safety.

We chose DyNetKAT [5] as it offers an analysable framework to specify the data and the control plane of SDNs. DyNetKAT uses NetKAT [2] as a model, and has some extensions such as synchronization, guarded recursion and multi-packet semantics, as seen in the syntax from Figure 4. NetKAT is a formal framework to specify and reason about networks [2]. NetKAT is a minimalist language with a denotational semantics based on Kleene Algebra with Tests (KAT) [16] succesfully used to model network packet flows. On the other hand, NetKAT does not fully depict some failures that can happen with SDNs. For example, one of the common uses of a SDN is to make a stateful firewall, however a stateful firewall cannot easily be programmed using NetKAT as it does not support dynamic changes.

While DyNetKAT can model a SDN with dynamic updates and can be used to prove correctness properties of a SDN, it cannot be used to explain and reason about the causes of safety violations and cannot be used to automatically suggest ways to avoid such safety violations. Correspondingly, this paper is about exploiting the operational semantics of DyNetKAT in order to reason about safety violations by means of causal reasoning.

Related Work. Multiple definitions on causality has been put out over the years, most of which made to be specific for the system it studies and the accompanying safety properties. This paper primarily relates to the works in [18] [11] [10] [20]. Our prototype makes use of the trace-based causal

reasoning for automata models proposed in [20], with the definition of causality accordingly with Halpern and Pearl. In order to conduct causal reasoning for SDNs, the framework specified in [5] has been adopted. These, along with the encoding with causality for labelled transition systems (LTS) proposed in [6], allowed us to compute causes in SDNs. Other works such as the NetKAT evaluation module of [24] used for LTS generation are referenced throughout the paper whenever applicable.

Our contributions. The following is a summary of the contributions of this paper:

- we provide, implement and reason about an algorithm to convert DyNetKAT into a Labelled Transition System (LTS) (Section 4.1) ; and
- we exploit the counterfactual causal reasoning theory on the generated DyNetKAT LTS to identify and explain designated safety failures (Sections 4.2 ,5).

Structure of the paper. In Section 2 we describe the two examples that will be focused on for LTS generating and causal reasoning purposes. In Section 3 we give a concise overview of NetKAT and DyNetKAT, explaining the syntax rules that were pivotal for this paper. We introduce and reason about a DyNetKAT formalism to LTS generation algorithm in Section 4.1. Following the LTS generation, we reason about how counterfactual reasoning theory can be exploited to compute causes in Section 4.2. We implement our cause computing method on the running examples as well as benchmarking the LTS generation on larger examples in Section 5 and report on its scalability. In Section 6, we wrap up the paper and suggest some directions for further research.

2 Running Examples

For the duration of this paper, we concentrate on modelling and cause computing for two realistic SDN instances that consists of a dynamically updated network configuration. The first example, Virtual Connection on Demand (VCoD) [12] application, a controller begins the update by setting up a virtual circuit between two hosts on the data plane. For the second one, a stateful firewall, a trusted host changes the state of the data plane by allowing a forbidden path in the network.

Example 1: A virtual circuit is created for the delivery of a bit stream between a source host and a destination host [22]. However, when necessary, another host should be able to send packages to the destination host provided that the destination host is not currently receiving a bit stream. An instance could be a user wanting to connect to an other host than the source host, and in that case if the virtual connection is not being utilized, the packages from the other host should reach the destination host.

Looking at the network topology depicted in Figure 1 , one could see that controllers *C1* and *C2* controls the network consisting of switches *S1* and *S2*. The virtual circuit exists between *H1* and *H3*, with the connection itself getting forwarded through the *port 1* and *port 2* of *S1*. When a virtual connection needs to initiated, *C2* is expected to notify *C1*. The connection between *H2* and *H3* is along *port 3* and *port 4* from *S2*. When *H2* wants to send something to *H3*, *S2* checks whether there is a virtual circuit between *H1* and *H3* by querying *C1*.

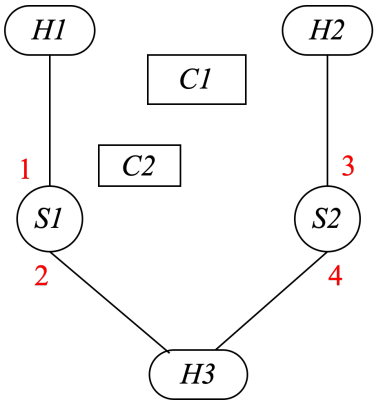

Fig. 1. Virtual Circuit

Example 2: This example is a modified version of the first running example of [5], used for testing and continuity purposes. It is about a stateful firewall designed to dynamically control the access to the intranet of an organization from the rest of the internet. When someone from the organization requests to receive packages from the rest of the internet, then the state of the firewall changes to permit packages by updating the flow tables. Correspondingly, a member of the organization can also revert the firewall to its initial state so to prevent the rest of the internet from accessing the organizations intranet.

Figure 2 depicts a condensed form of stateful firewall with the dotted line separating between the intranet of the organization and the internet. In the beginning, the switch *S1* forbids the transfer of packages from the internet to the intranet (*port 2* to *1*), while allowing packages from the intranet to the internet (*port 1* to *2*). The connection from the internet to the intranet (*port 1* to *2*) established when a request is sent from the Host *H1* to the Switch *S1*.

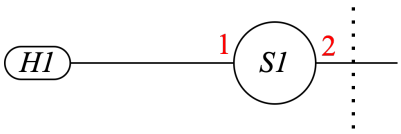

Fig. 2. Stateful Firewall

3 A brief overview of NetKAT and DyNetKAT

Here, we give a quick overview of the syntax and semantics of the DyNetKAT framework and the NetKAT language. As the DyNetKAT framework uses NetKAT as a base, it is necessary to have an understanding of NetKAT previous to learning about DyNetKAT. As a result of spacing concerns, the summary will only explain a chosen few from the syntax semantics, leaving the reader the option of referring to works in [5] and [2] if deemed necessary.

3.1 NetKAT overview

Definition 1 (*Network Packet*)**.** A packet is a function mapping field f_n from a set of fields with $f_n \in f_1, ..., f_k$ that map to a integer i from a set of integers $i \in 1, ..., k$ with a fixed integer k. We correspond $f_i(\sigma)$ with the value v_i in field f_i inside a packet σ. Using such a value v_i of valid type, we can either conduct a test or update the value of a function. For a test, we write $f_i(\sigma) = v_i$ which checks if the value of f_i from a packet σ is equal to v_i. In order to update a value in f_i from a packet σ we write $f_i(\sigma) := v_j$ which updates the value of f_i from a packet σ to v_j.

An empty list of such packets is denoted by $\langle\rangle$, correspondingly we represent a list l containing one packet σ with $\sigma :: \langle\rangle$. Prepending packet σ' to l would be denoted with $\sigma' :: l$, making position 0 of l the packet σ' and position 1 the packet σ.

The sequential composition operator $(p.q)$ seen in NetKAT syntax and semantics [2] in Figure 3 makes use of the Kleisli composition shown as $(p \bullet q)$. By way of explanation, it maps the inputted package using function specified by the left operand, and then maps the result using the function specified by the right operand.

NetKAT Syntax:

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

NetKAT Semantics:

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

Fig. 3. NetKAT: Syntax and Semantics [2]

3.2 DyNetKAT overview

DyNetKAT fundamentally extends NetKAT in three ways, synchronization, guarded recursion and multi-packet semantics. Synchronization allows for a basic handshake synchronization process and the ability to send a flow table in addition to the handshake. Guarded recursion is used for modelling state changes, however a policy has to be applied before a state change (guarding) to keep the formalism decidable. Multi-packet semantics are designed to treat a list of packets. We will summarize synchronization and guarded recursion in the following paragraphs.

The syntax of DyNetKAT [5] can be seen in Figure 4. Communication for synchronization in DyNetKAT is done with two phases. In the first phase, by using the operators $x!N; D$ or $x?N; D$, policies of type N are sent or received through channel x. Secondly, when the sending or receiving is successfully conducted, a new packet is fetched and processed in accordance with D.

$$\begin{array}{lcll} N & ::= & \text{NetKAT}^{-\mathbf{dup}} & \\ D & ::= & \bot \mid N; D \mid x?N; D \mid x!N; D \mid D \,||\, D \mid D \oplus D \mid X & (1) \\ & & X \triangleq D & \end{array}$$

Fig. 4. DyNetKAT Syntax [5]

Using these communication policies as a foundation, the operational semantics for synchronization can be seen in Figure 5. As visible, when both sending and receiving "agree" on a policy N and channel x, a reconfiguration based on policy N and channel x (`rcfg(x,N)`) can be observed.

$$(\mathbf{cpol}_{!?})\ \frac{(q, H, H') \xrightarrow{x!p} (q', H, H') \quad (s, H, H') \xrightarrow{x?p} (s', H, H')}{(q||s, H, H') \xrightarrow{\mathbf{rcfg(x,p)}} (q'||s', H, H')}$$

$$(\mathbf{cpol}_{?!})\ \frac{(q, H, H') \xrightarrow{x?p} (q', H, H') \quad (s, H, H') \xrightarrow{x!p} (s', H, H')}{(q||s, H, H') \xrightarrow{\mathbf{rcfg(x,p)}} (q'||s', H, H')}$$

Fig. 5. : Operational Semantics

While DyNetKAT allows for defining unguarded specifications, the authors in [5] build their frameworks on top of guarded specifications, where the occurrence of free variables are not allowed, as guardedness is assumed by default. To define recursive variable X acting in accordance to policy P , the defining equation $X \triangleq P$ is used.

In Equation 2, we provide a DyNetKAT formalism for the first running example, the VCoD application.

$$
\begin{aligned}
C1 &\triangleq NoVirtualCircuit!\mathbf{1}; C1 \oplus \\
&\quad VirtualCircuitReq?\mathbf{1}; C1' \\
C1' &\triangleq VirtualCircuitEnd?\mathbf{1}; C1 \\
C2 &\triangleq VirtualCircuitReq!\mathbf{1}; C2' \\
C2' &\triangleq VirtualCircuitEnd!\mathbf{1}; C2 \\
S1 &\triangleq VirtualCircuitReq?\mathbf{1}; S1' \\
S1' &\triangleq ((port = 1).(port \leftarrow 2)); S1' \oplus \\
&\quad VirtualCircuitEnd?\mathbf{1}; S1 \\
S2 &\triangleq NoVirtualCircuit?\mathbf{1}; S2' \\
S2' &\triangleq ((port = 3).(port \leftarrow 4)); S2 \\
Init &\triangleq C1||S1||S2||C2
\end{aligned}
\tag{2}
$$

As visible, when the formalization is initialized in $Init$, no virtual connection exists as $S1$ has no instructions about forwarding packages from *port 1* in its flow table. However, through sending $VirtualCircuitReq$ it can change its state into $S1'$, where instructions about forwarding packages from *port 1* exist. Similarly, $S2$ checks whether a virtual connection exists or not before forwarding packages from *port 3*, as it waits until receiving $NoVirtualCircuit$ to change its state to $S2'$, where it has forwarding instructions.

4 Methodologies

At the heart of this paper lies causality. In order to obtain the causes of violations, it is required to have the labelled transition system of the program that has the violation . A LTS is a transition system used to encode behaviour of programs, especially ones which work concurrently. To obtain the causes from the LTS, we will use an algorithm specified from the work in [20].

Definition 2 (*Labelled Transition System*)**.** A labelled transition system (LTS) is 4-tuple $(S, s_i, A, \rightarrow)$ in which S is a infinite set of states, $s_i \in S$) the initial state, A is an infinite set of actions and $\rightarrow \subseteq S \times A \times S$ is the transition relation.

4.1 LTS Setup, Parsing and Generation

The LTS generation is comprised of three stages, the *setup stage* for setting up the necessary environment and the structure of the DyNetKAT program, and the *parsing stage* for parsing the given DyNetKAT program, and the *generating stage* for generating an LTS from the given DyNetKAT program.

For the *setup stage*, the prototype tool implementation [1] behind the work in [5] was chosen as a base implementation. The implementation for this paper can be found at https://github.com/canolmezoglu/DyNetiKAT. Similarly to the prototype tool in [5], the LTS generator for this paper also uses a JSON type file [23] to define the inputted DyNetKAT expressions.

The JSON file used by the setup consists of five name/value pairs, `recursive_variables`, `channels`, `program`, `file_name` and `module_name`. Even though the protoype only needing a DyNetKAT formalism as input would have made it more intuitive to use, having an extra name/value pair defining the recursive variables i.e. `recursive_variables` prevents infinite rewriting of DyNetKAT's recursive variables. In addition, defining `channels` decreases ambiguity and potential syntax errors while parsing sending and receiving constructs. The definition of the program with `program` allows to define the starting point and the inputted packages in an intuitive way which is canonical with the introducing works from [5]. In order to define and generate error messages and Maude modules, `file_name` and `module_name` are used. A JSON file for a stateful firewall can be seen in Listing 1. Using the Maude parsing sub-module of the prototype from [5], the DyNetKAT JSON file is partly converted into a Maude System module for preventing issues such as infinite rewriting of DyNetKAT's recursive variables. An example of such module for the stateful firewall example can be seen in Listing 2.

Listing 1. JSON file depicting a DyNetKAT stateful firewal

```
{
  "module_name": STATEFUL-FIREWALL,
  "file_name": STATEFUL-FIREWALL.maude,
  "recursive_variables" :
  {
  "Switch" : (pt = 1 . pt <- 2) ; Switch o+ (pt =2
       . zero) ; Switch o+ (secConReq ? one) ;
      SwitchPrime,
  "SwitchPrime": (pt = 1 . pt <- 2) ; SwitchPrime
      o+ (pt = 2 . pt <- 1) ; SwitchPrime o+ (
      secConEnd ? one) ; Switch,
  "Host": (secConReq ! one) ; Host o+ (secConEnd !
       one) ; Host
  }
  "channels": ['secConReq', 'secConEnd'],
  "program": @Recursive(Switch)||@Recursive(Host),
       pt=01::pt=10::{},{},
}
```

Listing 2. Maude module to facilitiate parsing

```
load /src/maude/lts.maude

fmod STATEFUL-FIREWALL is
        protecting DNA .
        ops Switch SwitchPrime Host : -> Recursive .
        ops secConReq secConEnd : -> Channel .

endfm
```

[1] https://github.com/hcantunc/DyNetiKAT

During the *parsing stage*, we use the *Maude System* [7], a framework designed for equational and rewriting logic specification. The inputted DyNetKAT program is then rewritten into an identifier stating the type of the DyNetKAT expressions using the equational rewriting functions of Maude. The commutativity or associativity of DyNetKAT-specific constructs were represented using the *[⟨OperatorAttributes⟩]* components of Maude operator declarations. In order to make the parsing more performant by decreasing the state space for the Maude System, each program in `recursive_variables` is parsed separately. After parsing, the JSON file is updated by replacing of the programs in `recursive_variables` with the parsed programs. An example for the stateful firewall can be seen in Listing 3.

Listing 3. JSON file depicting a parsed DyNetKAT stateful firewall

```
{
  "module_name": STATEFUL-FIREWALL,
  "file_name": STATEFUL-FIREWALL.maude,
  "recursive_variables" :
  {
 "Switch": '@NetKAT((pt = 1 . pt <- 2)) ;
     @Recursive(Switch) o+ @NetKAT( (pt = 2 .
     zero)) ; @Recursive(Switch) o+ @Receive(
     @Channel(secConReq) >> one) ; @Recursive(
     SwitchPrime)',
 "SwitchPrime": '@NetKAT((pt = 1 . pt <- 2)) ;
     @Recursive(SwitchPrime) o+ @NetKAT((pt = 2 .
      pt <- 1)) ; @Recursive(SwitchPrime) o+
     @Receive(@Channel( secConEnd) >> one) ;
     @Recursive(Switch)',
 "Host": '@Send(@Channel(secConReq) >> one) ;
     @Recursive(Host) o+ @Send( @Channel(
     secConEnd) >> one) ; @Recursive(Host)'
  }
  "channels": ['secConReq', 'secConEnd'],
  "program": @Recursive(Switch)||@Recursive(Host),
       pt=01::pt=10::{},{},
}
```

As seen from *Init* in Equation 2, a DyNetKAT construct can consist of synchronized single recursive variables. Correspondingly, a `program` variable can contain single recursive constructs which can then contain more synchronized single recursive constructs inside, as visible in Figure 4. This presented a problem for the generation as the variables ahead of the variable being currently parsed would not be known. For instance, in a situation where the construct being parsed is of type sending, and there is a single recursive variable ahead which unfolds into receiving, a `RCFG(x,p)` would be missed unless every recursive variable is unfolded at each iteration making the generation a computationally intensive process. In order to avoid this problem, all of the single recursion inside a program is unfolded until it defines a policy that is not a single recursive variable by Algorithm 2 before the program gets evaluated by Algorithm 1 .

Throughout the *generation stage*, different methods of generation were used for the differing types of DyNetKAT constructs, as obtained from the *parsing stage*. The different methods were added in order to check the satisfaction of different conditions for the rules in the operational semantics in [5]. For a `RCFG(x,p)` transition, concisely defined as a synchronizing message passing through channel x, a linear search of receive messages of the same channel is conducted. NetKAT expressions are parsed according to the evaluation module in [24]. As the evaluation module is only accessible through an interactive shell of a spawned process, the querying for evaluation of NetKAT policies were automated by the *Pexpect* [25] package. The output packages might be differing with respect to cardinality, thereby we used arrays to represent them as states in the iterations of the generation method. With accounting for such differences, the final generation algorithm can be seen in Algorithm 1.

Algorithm 1 Algorithm for the generation phase

Input: A DyNetKAT formalism such as Equation 2
Output: A LTS G corresponding to the inputted formalism
initialize a list L;
append the *program* from the JSON into L;
initialize a LTS G;
while $|L| \geq 0$ **do**
  $curr \leftarrow L.pop()$;
  initialize lists P,H and $HPrime$;
  $P \leftarrow$ synchronized policies in $curr$ after being unfolded by Algorithm 2;
  $H \leftarrow$ list of the packages to process in $curr$;
  $HPrime \leftarrow$ list of processed packages in $curr$;
  for each policy pol in P **do**
    if pol has the non-deterministic choice operator **then**
      separate the non-deterministic policies in pol and run this algorithm on each one
    end if
    if pol is of type *sending* or *receive* **then**
      $new\_P \leftarrow P$ with pol modified as the next state;
      $new\_state \leftarrow curr$ with new_P as the policies;
      if $new\_state \notin G$ **then**
        append new_state to L;
      end if
      add the appropriate transition between $curr$ and new_state in G;
      if $(P \setminus pol)$ has at least one policy of type *receive* **and** pol is of type *sending* **then**
        $receive\_P \leftarrow$ the next states of the policies of type *receive*;
        $new\_P \leftarrow P$ modified with $receive\_P$ and next state of pol;
        $new\_state \leftarrow curr$ with new_P as the policies;
        if $new\_state \notin G$ **then**
          append new_state to L;
        end if
        add a RCFG transition between $curr$ and new_state in G;
      end if
    else
      $new\_H \leftarrow$ deep copy of H
      $N \leftarrow new\_H.pop()$;
      $processed\_N \leftarrow N$ processed with the NetKAT evaluator
      $processed\_N \leftarrow processed\_N$ merged with $HPrime$
      $new\_state \leftarrow curr$ with new_H as packages to process and $processed\_N$ as the processed packages;
      if $new\_state \notin G$ **then**
        append new_state to L;
      end if
      add a NetKAT transition between $curr$ and new_state in G;
    end if
  end for
end while

Algorithm 2 Algorithm to parse synchronized recursive policies

Input: Synchronized programs with recursive names which can unfold into more synchronized programs
Output: Synchronized programs that begin with communication or NetKAT constructs
initialize a list L;
for each synchronized program P **do**
  if P begins with type $recursive$ **then**
    initialize a list $unfolded$
    $unfolded \leftarrow$ the unfolded policies from P
    if $unfolded$ has synchronized programs **then**
      while a recursive program exists in $unfolded$ **do**
        unfold the recursion and add to $unfolded$
      end while
    end if
    merge $unfolded$ with L
  end if
  append P to L
end for
  return L

In Figure 6 we portray the generated LTS for the second running example, stateful firewall with one external packet to process, using methodology specified in Section 4.1. A succesful evaluation of a NetKAT policy N on packet σ with header $pt(\sigma) = 1$ leading to a transition labelled (σ, σ') with $pt(\sigma') = 2$ is depicted in the LTS as the expression seen below.

$$(N, \sigma :: \{\}, \{\}) \quad \xrightarrow{(\sigma, \sigma')} \quad (, \{\}, \sigma' :: \{\})$$

4.2 Causal Computation from LTS

The causality checking of DyNetKAT programs is based on the work in [20]. In [5], the authors consider LTSs as the natural semantic models of DyNetKAT. However, in [20], reasoning is done on Finite Automata (FA).

The difference between the definitions of a FA and a LTS explains how a LTS in our case can be converted into a FA.

Definition 3 (*Finite Automaton*)**.** A finite automaton is 5-tuple $(S, s_i, A, \rightarrow, F)$ in which S is a finite set of states, $s_i \in S$) the initial state, A is a finite set of actions, $\rightarrow \subseteq S \times A \times S$ is the transition relation, and F is the set of accepting states.

As visible from Definition 2 and 3, there is one difference that would matter in defining hazards and computing causes, FA's have accepting states while LTS's does not. As the states in a generated LTS all represent states that DyNetKAT can reach, and a DyNetKAT formalism can terminate at any given time, all of the states in the LTS are added to accepting states F when creating a FA. Aside from converting the generated LTS into a FA, the hazard also has to be encoded as a regular expression to be a valid input for [20].

In technical terms, the FA object is represented with a dot type file [15]. We chose to not use any specific libraries when the dot file was generated as we were unable to find a library that let the user add multiple edges with the same source and destination but with a different label. Correspondingly, we designed a basic generator that added the nodes,edges and labels from the generated LTS to a text file later converted into a dot file. This generated file in addition to an regular expression made from the LTS actions were inputted to the work in [20] to find the causes for the inputted effects.

Listing 4. Causal computation output of Running Example 1 (Excerpt)

```
graph search computed, len causes: 1404
minimal causes computed, len minimal causes: 2
cause: "RCFG( VirtualCircuitReq )"
cause: "VirtualCircuitReq !"
```

5 Evaluation

In this section, we provide the results of inputting the first running example to the process described in Section 4 and evaluate the performance of the algorithms described in Section 4.1 on various DyNetKAT formalisms.

A hazardous situation in the first running example can happen when *S1* is inside a virtual connection and *S2* also forwards a packet, leading *H3* to have a processing error. This hazard could be encoded in a regular expression, as seen with effect e in Equation 3.

$$e = (((port = 1).(port := 2)); (\neg VirtualCircuitEnd!)^*; \\ ((port = 3).(port := 4))) \tag{3}$$

As visible, when *S1* is inside a virtual connection, it will forward the packages with $port = 1$ to $port2$, thereby guaranteeing that in the beginning of Equation 3, S1 is a virtual circuit. Following that, $(\neg VirtualCircuitEnd!)^*$ states that the action can be anything but $VirtualCircuitEnd!$, repeated over any number of times. Consequently, before the last element of effect e , the virtual connection will not be terminated. The last element, $port = 3$ to $port4$, is the forwarding action *S2* does. If this action is done while a virtual connection still exists, as explained before it will lead *H3* to have a processing error.

The result of converting the LTS for running example one into a FA and putting that FA with Equation 3 into the causal reasoning tool from [20] can be seen in Listing 4.

It is straightforward from Listing 4 to see that 51 causes have been computed, *i.e* these 51 different sequences of actions, if activated, can produce the hazard described in Equation 3. We can deduce that the causes specified in the last two lines, "$RCFG(VirtualCircuitReq)$" and "$VirtualCircuitReq!$" do a better job at describing a potential sequence of actions as they are the *minimal* sequence. Consequently, we have seen that the hazard state can happen if *C2* sends a virtual connection request ("$VirtualCircuitReq!$"). Upon seeing this, it is clear that the error stems from a race condition between *S1* and S2 caused by *S1*'s request being received after *S2's* request.

To evaluate the performance of the LTS generation algorithm, we used a modified version of the FatTree [1] topology evaluation experiments conducted in [5] in conjunction with

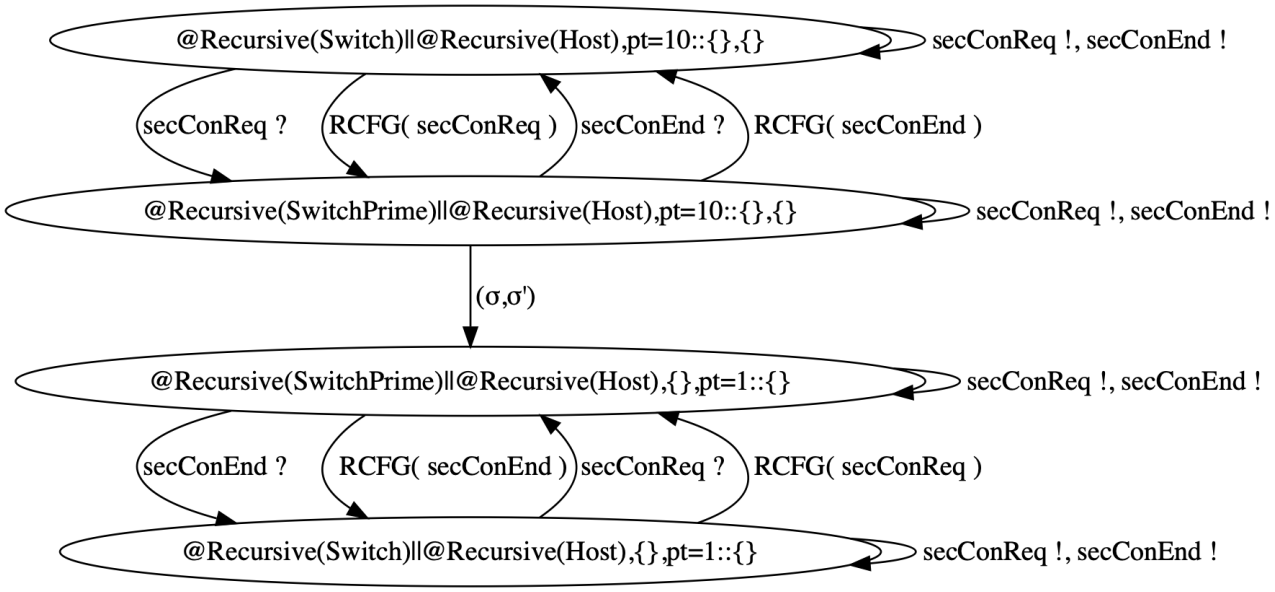

Fig. 6. Stateful Firewall LTS

some of the examples from the same paper. A FatTree consists of 3 levels, the core, the aggregate, and the top of the rack (ToR) with links between each switch and the switches in the level adjacent to them, as seen in Figure 7. We tested the generator on 3 FatTrees increasing in cardinality, with the largest one having 375 switches and 10 pods.

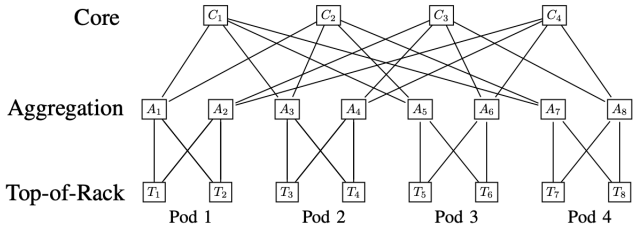

Fig. 7. A FatTree Topology [5]

The generated FatTree formalism for the experiment consists of two switches in different pods, which send packages between them through a firewall in an aggregation layer which then changes state. We put one package that will not be accepted through the firewall, generating a LTS simulating the alternative states and actions that happen.

The other formalisms consisted of the two running examples with two packages to process and the distributed independent controllers from [5] with one package that transferred between two switches. The computer used for the experiments had a 3.4Ghz Intel 6700 processor and 16 GB RAM, with Ubuntu 18.04 LTS as the operating system. The time shown in Figure 8 are time spent on the LTS generation, averaged over 3 runs.

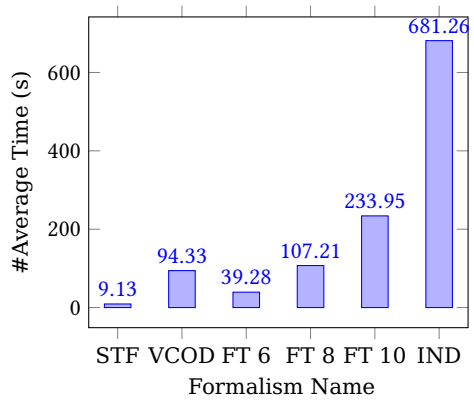

Fig. 8. Benchmarking Results

STF and VCOD are abbreviations for the two running examples respectively, while FT x is an abbreviation for a FatTree with x representing the number of pods, lastly IND is for distributed independent controllers example. These results indicate that the generating time is correlated with the amount of nodes or the number of states a DyNetKAT program can generate, and not the number of Pexpect calls made to evaluate NetKAT expressions. All of the FatTrees and the distributed independent controllers example have the same number of recursive variables at 16, nonetheless

the distributed independent controllers takes the most time in a margin as it has the highest number of states.

Further research into this benchmark confirms the correlation between the cardinality of the edges and nodes of the LTS with the time taken to generate it. As seen from Figure 9, the time increases with the number of states and edges of the LTS. However, despite the time doubling between generating the LTS for a FatTree with 8 pods and 10 pods, the edge number states the same. A closer investigation reveals that as the names of the states expand, as 10 pods require more complex flow tables and larger state names, the time increases after a certain threshold with regards to the amount of nodes are reached. The implementation uses a lot of hash table structures for understanding where some reconfiguration states happen, which leads to the checking whether certain states are already parsed or not taking a lot longer time. Thereby, the speed of the algorithm is mainly constrained by two factors, the amount of nodes and edges of the LTS and the length of the state names.

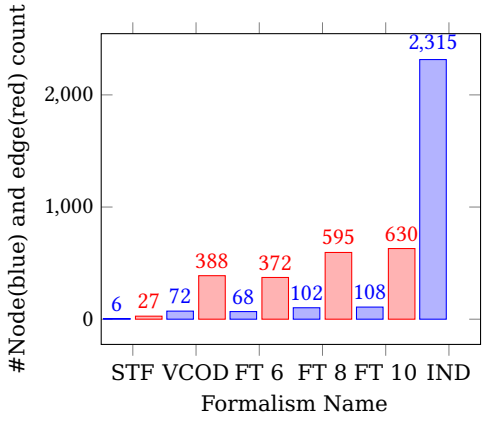

Fig. 9. The edge and node counts for the benchmarked examples [2]

6 Conclusions

We introduced a system for finding the causes of a hazard in a SDN. Our system builds upon the SDN framework DyNetKAT[5] and the causal reasoning algorithm specified in [20] and is consistent with the description of causation from [11] [10] in that hazards are only perceived when they are caused.

We only worked with hazards that could be defined using regular expressions composed of the action alphabet, thereby leaving room for potential improvements to check for hazards that occur after a certain state is reached or checking for the violation of liveness properties.

In order to adopt a trace-based system of causation according to the works of Halpern and Pearl [11] [10] and fit for the algorithm in [20], we created an LTS and converted it into a FA. During this implementation, we have made system to conduct basic lexical analysis on DyNetKAT using Maude, as well as doing some semantic analysis with Python to generate an LTS. Therefore, with some work being done, a DyNetKAT compiler with the target language set to be a SDN domain specific language such as P4 [3].

7 Acknowledgements

I would like to give special thanks to my supervisor Georgiana Caltais for her patience and consistent help throughout this research project. I also want to thank Hunkar Can Tunc for helping me understand the implementations for [5] and [20].

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Comput. Commun. Rev.* 38, 4 (aug 2008), 63–74. https://doi.org/10.1145/1402946.1402967

[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] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[4] Georgiana Caltais. 2019. Explaining SDN Failures via Axiomatisations. 303 (sep 2019), 48–60. https://doi.org/10.4204/eptcs.303.4

[5] Georgiana Caltais, Hossein Hojjat, Mohammad Mousavi, and Hunkar Can Tunc. 2021. DyNetKAT: An Algebra of Dynamic Networks. *arXiv preprint arXiv:2102.10035* (2021).

[6] Georgiana Caltais, Mohammad Reza Mousavi, and Hargurbir Singh. 2020. Causal Reasoning for Safety in Hennessy Milner Logic. *Fundamenta Informaticae* 173, 2-3 (2020), 217–251.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007. *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Vol. 4350. Springer.

[8] Holger Giese Gabor Karsai Edward Lee and Bernhard Rumpe Bernhard Schätz. 2007. Model-based engineering of embedded real-time systems. (2007).

[9] 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.

[10] Joseph Y. Halpern and Judea Pearl. 2013. Causes and Explanations: A Structural-Model Approach — Part 1: Causes. *CoRR* abs/1301.2275 (2013). arXiv:1301.2275 http://arxiv.org/abs/1301.2275

[11] Joseph Y Halpern and Judea Pearl. 2020. Causes and explanations: A structural-model approach. Part I: Causes. *The British journal for the philosophy of science* (2020).

[12] S Hares and M Chen. 2014. Use Cases for Virtual Connections on Demand (VCoD) and Virtual Network on Demand (VNoD) using Interface to Routing System. *Working Draft, IETF Secretariat, Internet-Draft draftthares-use-case-vn-vc-01. txt* (2014).

[13] David Hume. 1896. *A treatise of human nature*. Clarendon Press.

[14] Keith Kirkpatrick. 2013. Software-defined networking. *Commun. ACM* 56, 9 (2013), 16–19.

[15] Eleftherios Koutsofios and Stephen C North. 1996. Drawing graphs with dot. (1996).

[16] Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (1997), 427–443.

[17] W. S. Lee, D. L. Grosh, F. A. Tillman, and C. H. Lie. 1985. Fault Tree Analysis, Methods, and Applications A Review. *IEEE Transactions on*

Reliability R-34, 3 (1985), 194–203. https://doi.org/10.1109/TR.1985.5222114

[18] Florian Leitner-Fischer and Stefan Leue. 2013. Causality Checking for Complex System Models. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 248–267.

[19] David Lewis. 1973. Causation. *Journal of Philosophy* 70, 17 (1973), 556–567. https://doi.org/10.2307/2025310

[20] Hui Feng Hunkar Can Tunc Marcello Bonsangue, Georgiana Caltais. 2022. A Language-based causal model for safety. (2022).

[21] 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.

[22] Larry L Peterson and Bruce S Davie. 2007. *Computer networks: a systems approach*. Elsevier.

[23] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 263–273.

[24] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A fast compiler for NetKAT. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 328–341.

[25] Noah Spurrier. 2022. Pexpect version 4.8. https://pexpect.readthedocs.io/en/stable/. Accessed: 2022-05-29.