Counterfactual Causality in Networks

Georgiana Caltais and Can Olmezoglu

University of Twente, The Netherlands
g.g.c.caltais@utwente.nl , c.olmezoglu@utwente.nl

1 Introduction & Background

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 this abstract, we propose a framework for explaining violations of safety properties in Software Defined Networks (SDNs), using counterfactual causal reasoning [7].

SDN has gained a lot of traction due to its increased network management and programmability, achieved by decoupling the control plane from the data plane [8], in contrast with traditional networks. SDN technologies can play an important role in solving issues concerning big data applications, including data processing in cloud data centers, optimisations and data delivery. In this abstract, we focus on DyNetKAT –a rigorous framework for modelling and analysing (multi-)packet forwarding within an SDN, and communication between data and control planes– introduced in [3] by a subset of the authors. DyNetKAT is based on NetKAT [1], a minimalist language based on Kleene Algebra with Tests, supported by a sound and complete axiomatisation. Packets in (Dy)NetKAT are encoded as sets of fields and associated values $\{f_1 = v_1, \dots f_n = v_n\}$. NetKAT can model the forwarding of a single packet within a network, and includes constructs such as dropping of packets ($\mathbf{0}$), acceptance of packets ($\mathbf{1}$), multicast ($+$), packet fields modification ($f \leftarrow n$) and repeated application (*) of these policies. In addition, NetKAT can be used to build packet histories using the **dup** construct, which was dropped in [3]. DyNetKAT extends NetKAT with channel-based communication ($||$, $x?p$, $x!p$) of flow tables (p) between the data and control planes (with synchronous communication of p on channel x encoded as $\mathbf{rcfg(x, p)}$), no-behaviour policies ($\bot$), non-deterministic choice ($\oplus$), recursive specifications (X) and multi-packet semantics in the context of a sequential composition operator (;) that marks the fetching of a new packet in the packet queue. In contrast with NetKAT, DyNetKAT has an operational semantics that entails LTS models, and a sound and ground complete axiomatisation in the style of the Algebra of Communicating Systems (ACP). The syntax of DyNetKAT is: $N ::= \text{NetKAT}^{-\mathbf{dup}} \quad D ::= \bot \mid N \,;D \mid x?N\,;D \mid x!N\,;D \mid D\,||\,D \mid D \oplus D \mid X$ with $X \triangleq D$. The complete framework is defined in [3]. Similarly to [3], we consider guarded DyNetKAT specifications that can be reduced to equivalent expressions in head normal form (Lemma 7 in [3]). This, in turn, guarantees the existence of finite LTS models for DyNetKAT specifications with finite number of recursive variables, finite sets of channel names and packet fields over finite domains.

We base our SDN safety failure explanations on the so-called counterfactual causality, or actual causality, introduced in the seminal work [7], and adapted in [2] to the context of finite automata (as SDN models) and regular expressions (as a language for defining safety properties). Intuitively, (a sequence of) events c are considered causal with respect to the realisation of a hazard e whenever (i) c is necessary for e to happen, (ii) c not happening entails e not happening (this is known as the counterfactual test), (iii) there is no c' "simpler" than c that can satisfy the conditions above. In addition, there might be the case that despite c being observed, e does

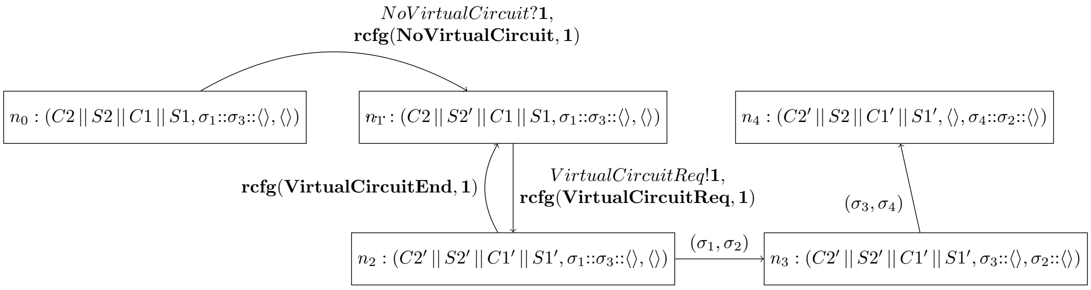
$NoVirtualCircuit?\mathbf{1}$,
$\mathbf{rcfg}(\mathbf{NoVirtualCircuit}, \mathbf{1})$
$n_0 : (C2 \,||\, S2 \,||\, C1 \,||\, S1, \sigma_1{::}\sigma_3{::}\langle\rangle, \langle\rangle)$
$n_1 : (C2 \,||\, S2' \,||\, C1 \,||\, S1, \sigma_1{::}\sigma_3{::}\langle\rangle, \langle\rangle)$
$n_4 : (C2' \,||\, S2 \,||\, C1' \,||\, S1', \langle\rangle, \sigma_4{::}\sigma_2{::}\langle\rangle)$
$\mathbf{rcfg}(\mathbf{VirtualCircuitEnd}, \mathbf{1})$
$VirtualCircuitReq!\mathbf{1}$,
$\mathbf{rcfg}(\mathbf{VirtualCircuitReq}, \mathbf{1})$
(σ_3, σ_4)
(σ_1, σ_2)
$n_2 : (C2' \,||\, S2' \,||\, C1' \,||\, S1', \sigma_1{::}\sigma_3{::}\langle\rangle, \langle\rangle)$
$n_3 : (C2' \,||\, S2' \,||\, C1' \,||\, S1', \sigma_3{::}\langle\rangle, \sigma_2{::}\langle\rangle)$

Figure 1: Virtual Circuit LTS (excerpt)

not happen due to some other cancelling actions (e.g., the forest does not burn down, despite the lightning, because the firefighters arrive on time). Such situations are modelled by means of contingencies in [7] or events causal by their non-occurrence in [2, 4]. The causal analysis in [2] is performed in the context of FA models, and safety violations, or hazards encoded as regular expressions defined in the standard fashion: $e \ ::= \ 0 \mid 1 \mid a \mid e\,;e \mid e + e \mid e^*$. The computed causes are words, or decorated traces $w_0\, a_0\, w_1\, a_1 \ldots a_n\, w_n$ where $a_0\, a_1 \ldots a_n$ is a word which, if executed, leads to the hazard e, and w_i ranges over contingencies that disable the hazard. Note that w_i play an important role in describing fixes, or alternative safe scenarios.

Our contribution. First, we devise and implement an algorithm that computes the LTS models of DyNetKAT programs. Then, we transform the aforementioned LTSs into FA models in a straightforward fashion, by handling every state as accepting. The generated FAs can be further analysed according to the causal inference machinery in [2]. We explain our approach based on a running example –a faulty virtual circuit that allows illegal packet forwarding–.

2 Running Example

A virtual circuit is created for the delivery of a bit stream between a source host and a destination host [10], denoted by $H1$ and, respectively, $H3$ in the example of Figure 2. However, when necessary, another host such as $H2$ in Figure 2 should be able to send external packets to $H3$, provided that $H3$ is not currently receiving a bit stream. Controller $C1$ oversees the network and sends messages to network devices $C2$, $S1$ and $S2$, deciding when packets from $H2$ are forwarded or whether a virtual circuit between $H1$ and $H3$ can be initiated. For example, if $H2$ wants to send something to $H3$, the switch $S2$ connecting $H2$ and $H3$ checks whether there is a virtual circuit between $H1$ and $H3$ by querying $C1$. When a virtual circuit needs to be initiated, $C2$ informs $C1$ to make sure other packets are not being sent while the virtual circuit is active. After receiving this information, $C1$ stops allowing external packets from $H2$ to be forwarded to $H3$. In equation (1) we provide a DyNetKAT formalism for the running example.

A hazardous situation in the running example can happen when a virtual connection between $H1$ and $H3$ is active and processing a packet σ_1 into σ_2, indicating switch $S1$ forwarding the package from port 1 to port 2, depicted in red color in Figure 2. As this forwarding operation is happening, a packet σ_3 from $H2$ is processed into σ_4 and forwarded to $H3$ by $S2$. As the circuit is already active and using most of the resources of $H3$, the arrival of σ_4 at $H3$ might lead to an overflow error.

The LTS behavioural model of the DyNetKAT program in Figure 2 can be devised according to the DyNetKAT operational semantics. An excerpt of this LTS is provided in Figure 1. For instance, the trace $\mathbf{rcfg}(\mathbf{NoVirtualCircuit}, \mathbf{1})$ $\mathbf{rcfg}(\mathbf{VirtualCircuitReq}, \mathbf{1})$ originating in n_0 leads to the state n_2 witnessing the hazard $h \triangleq ((\sigma_1, \sigma_2); (\neg VirtualCircuitEnd!\mathbf{1})^*; (\sigma_3, \sigma_4); A^*)$. The next goal is to exploit the causal machinery in [2] and derive causal explanations for safety failures in SDN in an automated fashion.

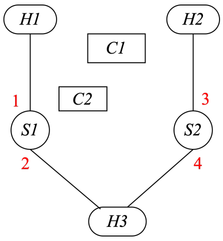

Figure 2: The Virtual Circuit

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

3 Methodology, Results and Extensions

The implementation for creating the LTS from DyNetKAT specifications is explained below and can be found at https://github.com/canolmezoglu/DyNetiKAT.

Methodology: To generate the LTS models, the prototype implementation [1] from [3] was chosen as the base implementation for parsing an inputted DyNetKAT specification. This implementation was modified using Maude [5] to classify different operators of DyNetKAT. Following the parsing, we implemented in Python an algorithm that exploits the operational semantics from [3] and extracts the LTS from the parsed specification. To obtain the causes from the LTS, we used Algorithm 1 from the work in [2], where the LTS was converted into a FA model by considering all the states of the LTS as accepting states.

Results: Upon conducting this methodology on the specification of the running example in Section 2, and using the regular expression h as the hazard, four causal explanations can be identified as (the minimal) traces leading from n_0 to n_2. These traces entail a race condition arising between the controllers when the virtual circuit was first made active. Note that, for the case study in this paper, there are no contingencies that can be used to steer the aforementioned causal explanations away from the undesired effect h. Hence, the actual causes coincide with the traces witnessing h in the LTS model of the virtual circuit in (1).

Extensions: Currently, we are working on developing a tool for extracting DyNetKAT specifications from real SDN data, based on the logs in [6] and OpenFlow [9]. The latter is a protocol that can manipulate the control logic of a network and program the flow table of network switches. As OpenFlow networks are working, or when they are simulated, all the modifications are stored in the form of logs, describing what flow table updates have been made by which controller and for which network devices. Using these logs, such as ones that could be obtained from the work in [6], the state changes of the network switches can be inferred and converted into DyNetKAT specifications. These specifications can be used for causal analysis on real world data, as well as benchmarking the current prototype implementation.

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

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: semantic foundations for networks. In Suresh Jagannathan and 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*, pages 113–126. ACM, 2014.

[2] Marcello M. Bonsangue, Georgiana Caltais, Hui Feng, and Hünkar Can Tunç. A language-based causal model for safety. In Yamine Aït Ameur and Florin Craciun, editors, *Theoretical Aspects of Software Engineering - 16th International Symposium, TASE 2022, Cluj-Napoca, Romania, July 8-10, 2022, Proceedings*, volume 13299 of *Lecture Notes in Computer Science*, pages 290–307. Springer, 2022.

[3] Georgiana Caltais, Hossein Hojjat, Mohammad Reza Mousavi, and Hünkar Can Tunç. Dynetkat: An algebra of dynamic networks. In Patricia Bouyer and 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*, volume 13242 of *Lecture Notes in Computer Science*, pages 184–204. Springer, 2022.

[4] Georgiana Caltais, Mohammad Reza Mousavi, and Hargurbir Singh. Causal reasoning for safety in hennessy milner logic. *Fundamenta Informaticae*, 173(2-3):217–251, 2020.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. The maude system. In Paliath Narendran and Michaël Rusinowitch, editors, *Rewriting Techniques and Applications, 10th International Conference, RTA-99, Trento, Italy, July 2-4, 1999, Proceedings*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243. Springer, 1999.

[6] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin T. Vechev. Sdnracer: concurrency analysis for software-defined networks. In Chandra Krintz and 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*, pages 402–415. ACM, 2016.

[7] Joseph Y. Halpern and Judea Pearl. Causes and explanations: A structural-model approach — part 1: Causes. *CoRR*, abs/1301.2275, 2013.

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

[9] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. Openflow: enabling innovation in campus networks. *Comput. Commun. Rev.*, 38(2):69–74, 2008.

[10] Larry L. Peterson and Bruce S. Davie. *Computer networks - a systems approach (3. ed.)*. Morgan Kaufmann, 2003.