University of Sussex

ABSTRACTIONS AND OPTIMISATIONS FOR MODEL-CHECKING SOFTWARE-DEFINED NETWORKS

by

Vasileios Klimis

Submitted for the degree of Doctor of Philosophy

University of Sussex

Declaration

I declare that (1) all the work contained herein is my own and no work is unacknowledged, and (2) that this work has not been and will not be submitted in whole or in part for the award of any other degree.
This thesis contains published/accepted work and work prepared for publication.

Signature:

VASILEIOS KLIMIS

Abstract

Vasileios Klimis, Ph.D.

ABSTRACTIONS AND OPTIMISATIONS FOR MODEL-CHECKING SOFTWARE-DEFINED NETWORKS

Software-Defined Networking introduces a new programmatic abstraction layer by shifting the distributed network functions (NFs) from silicon chips (ASICs) to a logically centralized (controller) program. And yet, controller programs are a common source of bugs that can cause performance degradation, security exploits and poor reliability in networks. Assuring that a controller program satisfies the specifications is thus most preferable, yet the size of the network and the complexity of the controller makes this a challenging effort.

This thesis presents a highly expressive, optimised SDN model, (code-named MoCS), that can be reasoned about and verified formally in an acceptable timeframe. In it, we introduce reusable abstractions that (i) come with a rich semantics, for capturing subtle real-world bugs that are hard to track down, and (ii) which are formally proved correct. In addition, MoCS deals with timeouts of flow table entries, thus supporting automatic state refresh (soft state) in the network. The optimisations are achieved by (1) contextually analysing the model for possible partial order reductions in view of the concrete control program, network topology and specification property in question, (2) pre-computing packet equivalence classes and (3) indexing packets and rules that exist in the model and bit-packing (compressing) them.

Each of these developments is demonstrated by a set of real-world controller programs that have been implemented in network topologies of varying size, and publicly released under an open-source license.

To my mother

Acknowledgements

I would like to thank both my supervisors, Bernhard Reus and George Parisis for their undiminished enthusiasm, encouragement, time and support, and for getting me to accomplish more than I ever thought possible.

I would like to thank the School of Engineering and Informatics at the University of Sussex, for awarding me a full-time 3-year scholarship to fund this PhD project.

My gratitude extends to StudyGroup and especially to Agneau Belanyek for keeping this PhD financially viable.

Gratitudes go to my lab mate, Mohammed Alasmar, too, for the fun-time we had together playing table tennis: without him this PhD would have been completed in half the time.

My big special mentions go to my shadow supervisor, my wife, for her patience: she motivated me by constantly asking: when will you finish that #&!* PhD?

Contents

List of Chapters Published as Papers in Peer-Reviewed Conferences ix

List of Tables x

List of Figures xi

List of Controller Programs xii

Abbreviations and Acronyms xiii

Nomenclature and Notations xiv

OpenFlow Messages and the Respective Modelled Actions in MoCS xviii

1 Introduction 1

1.1 Thesis Contributions 2

1.2 Thesis Overview 3

2 Background: A Survey of Computer Network Verification Approaches 6

3 Towards Model Checking Real-World Software-Defined Networks 30

4 Model Checking Software-Defined Networks with Flow Entries that Time Out 57

5 Conclusions and Future Work 65

5.1 Future Directions 65

5.2 A final remark 67

Extended Bibliography 68

A Artifact for Paper: "Towards Model Checking Real-World Software-Defined Networks" 96

List of Chapters Published as Papers in Peer-Reviewed Conferences

Chapter 3 Klimis V., Parisis G., Reus B.: Towards Model Checking Real-World Software-Defined Networks. In: Computer Aided Verification, CAV 2020. Lecture Notes in Computer Science, vol 12225, pp 126–148. Springer, Cham. https://doi.org/10.1007/978-3-030-53291-8_8

Chapter 4 Klimis, V., Parisis, G., Reus, B.: Model Checking Software-Defined Networks with Flow Entries that Time Out. In: Formal Methods in Computer-Aided Design, FMCAD 2020.

List of Tables

Chapter 3

1 Safeness Predicates 42

Chapter 4

1 Performance by number of clients and servers 62

A

1 Memory usage and verification runtimes 99

2 Dataplane topologies 106

List of Figures

Chapter 1

1 Schematic outline of the thesis 5

Chapter 2

1 Pictorial representation of the main types of network correctness properties 10

2 The queues and connections of the KUAI model 11

3 The Inverse Transfer Function 14

4 Example of a concolic execution in NICE 18

Chapter 3

1 A high-level view of MoCS 34

2 An example run in MoCS 39

3 Packet and rule indices 44

4 Performance Comparison – Verification Throughput 46

5 Performance Comparison – Visited States 47

6 Performance Comparison – Memory Footprint 47

7 Two networks 49

Chapter 4

1 A high-level view of extended MoCS 59

2 Four clients and two servers connecting to an OF- switch 60

3 The causal enabling relation between actions for an additional packet 61

4 Explored States in extended MoCS 62

List of Controller Programs

Chapter 3

CP 1: A stateless firewall with control messages reordering bug 54

CP 2: Stateful inspection firewall 55

CP 3: MAC learning application: for verifying absence of loops 55

CP 4: Wrong nesting level bug 56

CP 5: Consistent updates. 56

Chapter 4

CP 1: Packet-In Message Handler 62

CP 2: Naive Flow-removed message handler 62

CP 3: Correct Flow-removed message handler 62

Abbreviations and Acronyms

MoCS . . . . . **Mo**del **C**hecking for **S**oftware-Defined Networks

SDN . . . . . Software-Defined Networking

MC . . . . . Model Checking

POR . . . . . Partial-Order Reduction

LTL . . . . . Linear-Time Temporal Logic

CP . . . . . Controller Program

SSH . . . . . Secure Shell

OF . . . . . OpenFlow

CTX . . . . . Context

TCP . . . . . Transmission Control Protocol

Nomenclature and Notations

$Hosts$	the set of all hosts in the network
$Switches$	the set of all switches in the network
n	node: a network device (either host or switch): $n \in (Hosts \cup Switches)$
$Ports(n)$	the set of ports of node n: $Ports(n) \subseteq \mathbb{N}$
pt	port in node n: $pt \in Ports(n)$
$ports$	a subset in $Ports(n)$
loc	location: $loc = (n, pt)$
Loc	the set of all locations
λ	the network topology: a bijection which associates a physical network interface (a location) with another one. Formally, $\lambda : Loc \rightarrow Loc$
$Packets$	the set of all packets in the network
pkt	packet: a tuple of (1) a set of abstract (proof-relevant) packet matching header fields and (2) a location loc
$Barriers$	the set of all barrier IDs: $Barriers \subseteq \mathbb{N}$
b	barrier: $b \in Barriers$
r	rule: a tuple of ($priority$, $pattern$, $ports$, $timeout$), where $priority \in \mathbb{N}$ and $pattern$ is a match condition over the header fields of packets defining so a set of packets $packets \subseteq Packets$.
$Rules$	denotes the set of all rules (flow entries)
$rcvq$	receive queue
pq	packet queue
rq	request queue

fq	forward queue
cq	control queue
brq	barrier-reply queue
frq	flow-removed queue
ft	flow table: $ft \subseteq Rules$
$head(q)$	the element with the "longest time" in the queue q
h, ...	host (client or/and server): $h = (ports, rcvq) \in Hosts$
c, ...	client: $\mathtt{c} \in Hosts$
s, ...	server: $\mathtt{s} \in Hosts$
sw, ...	switch: $sw = (ports, ft, pq, cq, fq) \in Switches$
$pktIn$	handler for the OpenFlow `Packet-In` message
$barrierIn$	handler for the OpenFlow `Barrier Reply` message
$flowRmvd$ . . .	handler for the OpenFlow `Flow-removed` message
CP	controller program: CP $= (pktIn, flowRmvd, barrierIn)$
CS	the set of all controller program states
cs_0	the initial controller program state ($cs_0 \in CS$)
cs	the current controller program state ($cs \in CS$)
π	an assignment to host's receive queue, i.e., $\pi : Hosts \rightarrow \{rcvq\}$
δ	a function which maps each switch to its buffers. Formally, $\delta : Switches \rightarrow \{pq, fq, cq, ft\}$
γ	the current controller state which consists of the controller program state and the states of rq, brq, frq, i.e., $\gamma = (cs, rq, brq, frq)$
$controller$, ▲ .	SDN controller: $controller = (CS, cs_0, \gamma, \text{CP})$
S	the state-space of the overall system
s	a state in S: $s = (\pi, \delta, \gamma) \in S$
s_0	the initial state of the overall system: $s_0 \in S$
$\alpha(\cdot)$	a parametrised action
$Send$	the set of all $send(\cdot)$ actions
$Recv$	the set of all $recv(\cdot)$ actions

$Match$	the set of all $match(\cdot)$ actions
$NoMatch$	the set of all $nomatch(\cdot)$ actions
$Ctrl$	the set of all $ctrl(\cdot)$ actions
Add	the set of all $add(\cdot)$ actions
Del	the set of all $del(\cdot)$ actions
Fwd	the set of all $fwd(\cdot)$ actions
$Brepl$	the set of all $brepl(\cdot)$ actions
$Bsync$	the set of all $bsync(\cdot)$ actions
$Frmvd$	the set of all $frmvd(\cdot)$ actions
$Fsync$	the set of all $fsync(\cdot)$ actions
A	the set of all actions: $A = Send \cup Recv \cup Match \cup NoMatch \cup Ctrl \cup Add \cup Del \cup Fwd \cup Brepl \cup Bsync \cup Frmvd \cup Fsync$
$A(s)$	the set of all enabled actions in state s
$\hookrightarrow$	the transition relation
$s \xhookrightarrow{\alpha(\vec{a})} s'$	we say s enables $\alpha(\vec{a})$, where $\vec{a}$ are the arguments the guards which are satisfied by s are referring to
CTX	a context CTX = (CP, λ, φ)
AP	the set of atomic propositions
L	a labelling function which relates to any state $s \in S$ a set $L(s) \in 2^{AP}$ of those atomic propositions that are true for s
$\mathcal{M}_{(\lambda,\mathbf{CP})}$	a model parametrised by (1) the underlying data-plane *topology* λ, and (2) the *controller program* CP in use
t_i	projects the i-th co-ordinate of the tuple $t = (x_1, x_2, \ldots x_n)$
$s.\pi.n.q$	refers to the queue q of node n in state s (a dot notation to directly access nested functions and immutable fields in a tuple).
$\text{LTL}_{\setminus\{\bigcirc\}}$	the set of all LTL formulas without “next-step" operator $\bigcirc$
φ	LTL formulae over AP
$\models$	satisfaction relation
$Paths(s)$	the set of all paths starting in state s
π	an initial path (run) as a transition sequence $s_0 \xhookrightarrow{\alpha_1} s_1 \xhookrightarrow{\alpha_2} \ldots$

$trace(\pi)$ . . . . . trace of a path π, notated also as $L(s_0)L(s_1)\dots L(s_i)\dots$

$Traces(\mathcal{M})$ . . . the set of all traces of the initial states of $\mathcal{M}$

$s \models \varphi$ . . . . . state s satisfies the formula φ, i.e., the evaluation induced by $L(s)$ makes φ true. Formally, $s \models \varphi$ iff $L(s) \models \varphi$

$trace(\pi) \models \varphi$ . . the trace of path π satisfies LTL formula φ, i.e., for every state s_i in π, the evaluation induced by $L(s_i)$ makes φ true

$Traces(\varphi)$ . . . . the set of all infinite words (traces) σ over the alphabet 2^{AP} induced by an LTL formula φ, i.e., $Traces(\varphi) = \{\sigma \in (2^{AP})^{\omega} \mid \sigma \models \varphi\}$

P . . . . . specification property as a set of traces induced by an LTL formula φ i.e., $P = Traces(\varphi)$

$P(pkt)$ . . . . . denotes a predicate on packet *pkt* encoding a property of *pkt* based on its header fields.

$\mathcal{M} \models \varphi$ . . . . . model $\mathcal{M}$ satisfies φ, i.e., all its traces (behaviours) are admissible. Formally, $Traces(\mathcal{M}) \subseteq Traces(\varphi) = P$

$\mathcal{M} \overset{\mathsf{st}}{\equiv} \mathcal{M}'$ . . . . stutter-trace equivalence: for each path in $\mathcal{M}$ there exists a stutter-trace equivalent path in $\mathcal{M}'$, and vice-versa

$[\alpha(\vec{x})]P$ . . . . a proposition which is evaluated to true iff, after firing action $\alpha(\vec{a})$, P holds with the variables in $\vec{x}$ bound to the corresponding values in the actual arguments

OpenFlow Messages and the Respective Modelled Actions in MoCS

OpenFlow Message	Modelled Actions	Description	
Packet-In	$nomatch$	For packets sent from the switch to the controller	Asynchronous
Flow-removed	$frmvd$	Sent by the switch to the controller when a flow entry is removed from the flow table	
Packet-out	$\begin{pmatrix} ctrl \\ bsync \\ fsync \end{pmatrix} \times (fwd)$	Used by the controller to send a packet out of a specified port of the switch	Controller-to-Switch
Flow Mod	$\begin{pmatrix} ctrl \\ bsync \\ fsync \end{pmatrix} \times \begin{pmatrix} add \\ del \\ mod \end{pmatrix}$	Used to add/delete/modify flow entries	
Barrier Request	$\begin{pmatrix} ctrl \\ bsync \\ fsync \end{pmatrix} \times (b)$	Used to ensure message dependencies	
Barrier Reply	$brepl$	Sent by the switch to the controller after the switch completes processing for all operations requested prior to the Barrier Request message	

Chapter 1

Introduction

Computer networking is one of the most significant and fastest-growing developments of our age. At the time of the author's adolescence, in the late 80's, the Internet was still in its infancy: an insider-only academic/military experiment. Three decades later, the Internet has become almost indispensable.

Networks are of an increasingly disaggregated nature. As they have grown in size, ubiquity and importance – and our demands upon them are becoming so far-sighted – their complexity has also grown in line, ratcheting up unforeseen flaws and vulnerabilities. In order to meet the demands of such transformation, networks must become simpler to run. Achieving simplicity through software has generally proven to be an effective strategy.

Software-Defined Networking (SDN) enables networks to be managed through software. It centralises the programmability and management of the distributed network by abstracting the network's control logic away from the underlying physical devices through a software abstraction layer on a centralized controller. This allows networks to run on open standards (such as the OpenFlow protocol) and bare-metal hardware. Being free from proprietary lock-in creates more flexibility, interoperability and automation for networking devices.

Contradictorily, the advantages of this programmatic framework are the source of additional challenges to be mindful of. Decoupling the control plane from the data plane, introduces new surface areas such as the SDN controller, its protocols and network function APIs to attack: the more software runs a network, the more vulnerabilities you are exposed to, and inevitably the more bugs and exploits. Further, using an open source controller and network function applications can be tricky and dangerous as open source software is an attractive target for attackers and, to a certain extent, less secure. With all this challenge, now more than ever we need verification approaches that can (1) truthfully

capture and represent the behaviour of interest of the system and (2) automatically analyse the resulting behaviour model and determine in a reasonable amount of time whether the behaviour of the system is among the set of behaviours that are allowed by the desired correctness property.

The usual practice of checking the correctness of networks was largely and for too long (and still is) based on unsystematic best-effort/best-guess approaches. Formally reasoning about networks requires constructing models that reflect commonly exhibited behaviour. However, the state-of-the-art approaches suffer not only from the lack of model generality in describing faithfully the world, they often lack even a precise and unambiguous specification language for expressing the intended behaviour. As a result, subtle flaws may go undiscovered.

The thesis investigated in this work is that **scalable formal verification techniques based on equivalences and abstraction, accustomed to the domain of Software-Defined Networks, allow properties to be addressed using model checking**.

Formal techniques have not seen widespread adoption in Software-Defined Networks' verification due to the scale of the problems of interest. This thesis takes a pragmatic turn towards verifying properties of real-world Software-Defined Networks. It does so by proposing a formal foundation for network reasoning: a highly expressive, yet optimised, OpenFlow/SDN relational model which can represent network behaviour more realistically and verify larger deployments using fewer resources.

1.1 Thesis Contributions

The core contributions of this work are in devising domain-specific abstraction techniques that create smaller, high-level (abstract) models, allowing formal reasoning to scale up to the problems arising in SDN verification. The novelty comes not from the reductions, but instead, from their *contextual* adaptation to the problems at hand.

Properties, in this dissertation, are checked using model checking. Unfortunately, current model checking engines are unable to scale up to handle problems as large as those arising in naïve formulations of SDN behaviours. In this dissertation, scalable model checking of Software-Defined Networks is achieved using techniques such as abstraction and optimisation, but applied in a way that is well-matched to the problems that arise in SDNs. Chapters 3 and 4 address the challenge of scalability by employing abstraction mechanisms that are based on stutter-trace equivalence for avoiding redundant executions. The abstractions work in customised ways based on the *context* of their inputs; this can apply

either to a network topology, controller program or property.

Another contribution of this work is the use of model checking with flow entries that discretely time out (in Chapter 4). This enhancement considers logical timeouts that can be modelled either as random discrete ones, which means that installed rules which are flagged as 'timeout-removable' can be removed at any time, (as demonstrated in Chapter 4), or as timeouts which are bounded by integers.

1.2 Thesis Overview

The remainder of the thesis is structured as follows; Chapter 2 presents a taxonomic literature review which critically highlights the state of research on network verification. We also review previous work upon which our research draws. The survey (prepared as manuscript) is being planned for publication shortly.

Chapter 3, presents the core work of this thesis. Overall, the approach is based on model checking selecting representatives per-context from the equivalence classes of behaviours. In it, we (i) describe a rich interleaving model of concurrency for asynchronous systems to capture complex interactions between the SDN controller and the underlying network, (ii) present an expressive and well-defined specification language for specifying the correct behaviour of SDNs, (iii) propose context-aware (partial order) optimisations exploiting the commutativity of concurrently executed transitions by relevant contexts in which • the concrete *control program*, • the underlying data-plane *topology* and • the *specification* in question appear; the aim is to diminish as far as possible the size of the state space that needs to be searched, improving thus the performance of model checking, (iv) propose state representation optimisations, namely ▪ packet and rule indexing, ▪ identification of packet equivalence classes and ▪ bit packing, to improve performance, (v) establish the soundness of the proposed optimisations, and (vi) demonstrate the superiority of our model and specification language compared to the state-of-the-art in terms of model expressivity and performance/scalability (verification throughput and memory footprint); to demonstrate the applicability of the proposed approach, we perform extensive experiments on several popular benchmarks and real-world applications.

Properties To specify properties of packet flow in the network, in Chapter 3 we define an expressive specification language which provides the semantics necessary to describe the intended behaviour precisely. We use LTL formulas without "next-step", which allows capturing temporal relationships, and define the shape of the atomic state propositions so that they can be unambiguously interpreted. Using this logic, we can express properties

such as connectivity/reachability/isolation between(of) sets of nodes/ports, access control (black/while listing), in-order delivery, loop-freedom, waypointing, lack of blackholes, non-bypassability, routing with hops constraints, to name a few.

In Appendix A, we describe in detail a list of artifacts for our model (source code, topological diagrams, and use cases) and demonstrate how to reproduce the experimental results from the paper in Chapter 3 using the artifacts.

In Chapter 4 we present an extension of our model to deal with timeouts of flow table entries, and thus complying with the OpenFlow specification[1]. By modelling forwarding states that expire, it is possible to explore elusive aspects of 'disconnected' states between control and data plane, i.e., stale states in which the controller does not reflect the underlying data changes. We also propose optimisations that are customised to this extension and provably preserve intended correctness properties. We evaluate the performance of the proposed model in terms of verification performance and scalability using a load balancer and firewall controller program combo in network topologies of varying size.

[1] https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

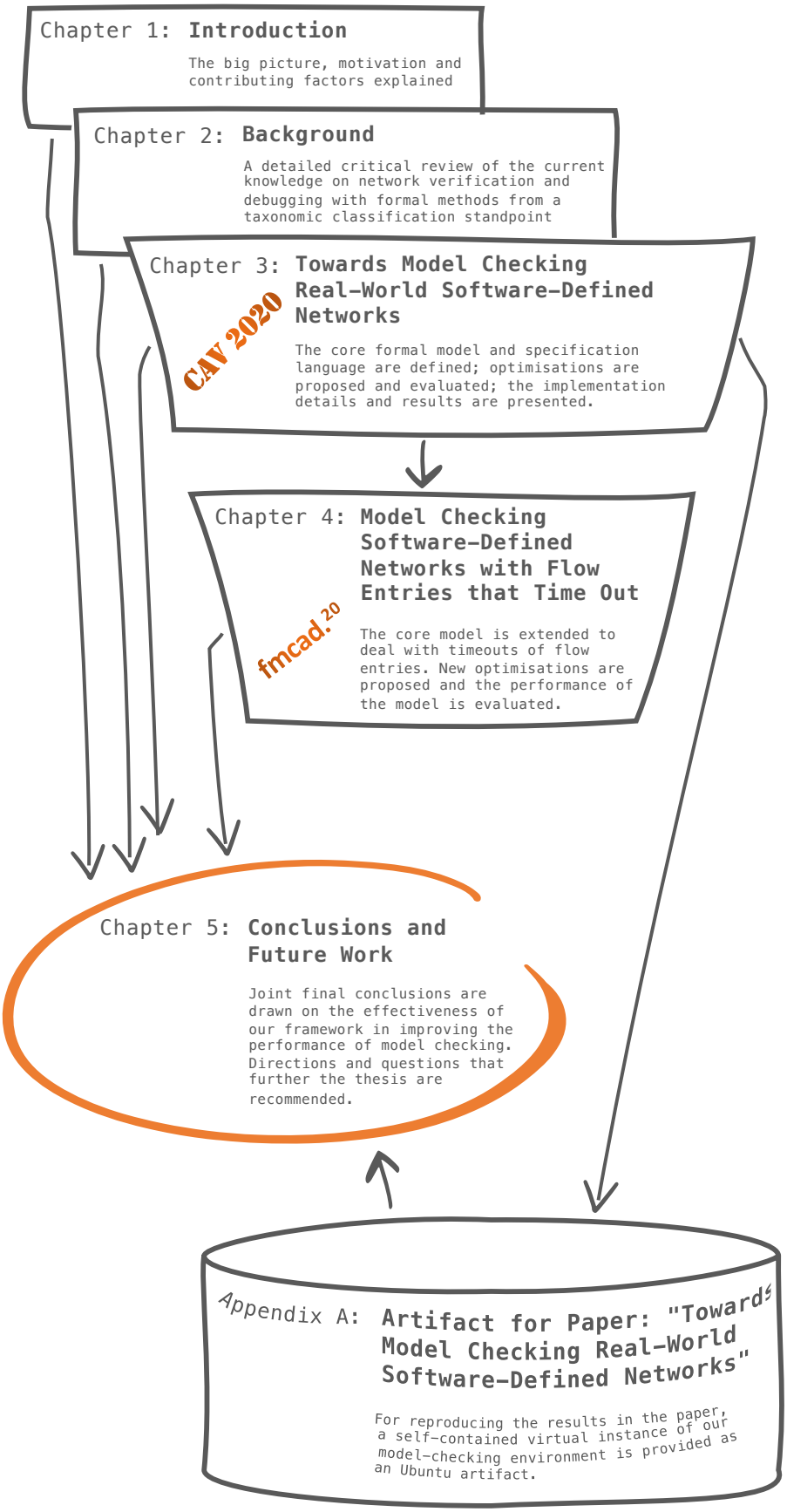


Fig. 1: Schematic outline of the thesis

Chapter 2

Background: A Survey of Computer Network Verification Approaches

This chapter sets the initial groundwork and the context required to situate our research contributions within the agenda of network verification. Here we begin by introducing a classification of the network correctness properties. We then provide a taxonomy of network verification approaches and analyse the literature relevant to the concerns of this thesis through the prism of this taxonomy. The aim of this comprehensive and critical review is to draw clear links between different verification approaches for computer networks.

A Survey of Computer Network Verification Approaches

Vasileios Klimis, George Parisis and Bernhard Reus
University of Sussex, UK
{v.klimis, g.parisis, bernhard}@sussex.ac.uk

***Abstract*—Computer networks keep growing in size, functionality and complexity leading to a need for powerful network segmentation and abstractions. While Software-Defined Networking brings some order to managing network complexity, its software stack is complex, highly asynchronous and distributed. This enormous amount of state yields an additional degree of complexity that traditional troubleshooting methods have become inefficient to deal with the entire system spectrum.**

In this review we provide a detailed critical overview of verification approaches that have been used thus far for computer networks, through the lens of a taxonomic classification.

***Index Terms*—SDN, Network Verification, Formal Methods, Model Checking**

I. Introduction

Traditional networks are built on closed systems that support a mixture of open and proprietary network protocols. Operating and extending the functionality of such networks is far from straightforward, as the network's control and data planes are intertwined within the devices themselves. Control and data plane functionality is controlled and can only be extended/updated by device manufacturers, who are also responsible for verifying the correctness of network hardware and software. Although this results in well-tested systems (with the caveat of slow update cycles and reluctance to innovation), networks are distributed in nature, therefore verifying that they operate as intended is very challenging. Network testing and debugging remains largely an unsystematic and error-prone process that is supported by tools developed decades ago, such as `SNMP`, `traceroute`, `tcpdump` and `netflow` [1]. Techniques that rely on collecting and analysing snapshots of network configuration to diagnose specific types of network problems have been developed, however debugging modern complex network deployments remains an open research challenge.

Software-Defined Networking (SDN)[1] [3]–[5] has brought about a paradigm shift in designing and operating computer networks. The key premise of SDN is the clean separation between the network control and data planes. With OpenFlow, a logically centralised controller implements the control logic, which is responsible for 'programming' the data plane. Communication between the controller and network devices is supported by Openflow [6], a standardised protocol, which is accessed through the *southbound* API [7]. Non-standard *northbound* APIs [7] expose higher-level control functionality to network programmers. The data plane is defined by flow tables that can be manipulated by the SDN controller through the southbound API. Recently, the P4 language [8] enabled protocol-independent packet processing that supports reconfigurability of packet processing at network devices.

The unified vision is one where controllers explicitly program network devices rather than assuming fixed switch designs and static header processing. SDN enables the rapid development and deployment of advanced and diverse network functionality. It has been employed in designing next-generation inter-data centre traffic engineering [9]–[15], load balancing [16]–[22], firewalls [23]–[27], Internet exchange points (IXPs) [28], optical [29]–[33], and home networks [34], [35]. Although SDN has been considered predominantly in wired networks and data centres settings, it has also been introduced in other environments, though being in the early stage for real networking, including wireless networks [36]–[48], wireless sensor networks [49], wireless mesh networks [50], infrastructure-less networks such as mobile/vehicular *adhoc* networks (MANET/VANET) [51], [52]. SDN is a key driver of network function virtualisation, edge computing and seamless virtual-machine migration.

SDN has gained noticeable ground in the industry, with major vendors supporting OpenFlow in their products (e.g. Hewlett-Packard and NEC were the first to integrate OpenFlow). SDN has been deployed at scale; e.g. Google's B4 deployments [53], Microsoft Azure cloud computing platform [54], Nicira's Network Virtualization Platform [55] and NTT's OpenFlow-based Gateway [56]. Large cloud providers, network operators and vendors have joined SDN industry consortia, such as the Open Networking Foundation [57] and the Open Daylight initiative [58]. P4 is also gaining traction; it has been recently combined with the Open Networking Foundation (ONF) and the Linux Foundation.

In network verification one distinguishes between data-plane verification and control-plane verification. Networks forward packets according to routing tables switches, the network's data plane (or forwarding plane). Accordingly, verification should cover properties regarding packet propagation and packet integrity. Networks generate those routing tables through protocols such as BGP (border gateway protocol). Accordingly, network verification covers properties regarding the control plane, the correct setup of the routing tables. We distinguish pre- (and post-) deployment verification as well as bug detection. Verification of the control plane shares many

[1] Although the concept of network programmability has been much debated for as long as computer networks itself, the term software-define networking was first coined in [2].

issues with traditional program verification while verification of the data plane shares many traits with reachability analysis in graphs.

SDN presents a unique opportunity for innovation and rapid development of complex network services by enabling all players, not just vendors, to develop and deploy control and data plane functionality in networks. This comes at a great risk; deploying buggy code at the control and/or data plane, and problematic flow entries at the data plane would potentially result in network and service disruption and security loopholes. Understanding and fixing such bugs is far from trivial, given the distributed nature of computer networks, the complexity of the control plane and the concurrency present in networks.

In this paper we present a comprehensive and critical review of verification approaches for computer networks. We (1) describe and classify local and network-wide properties, the correctness of which is verified by said approaches; (2) provide a taxonomy of network verification approaches; (3) analyse the relevant literature through the lens of the proposed taxonomy; (4) provide a critical analysis of the key research challenges that require further attention by the community. To the best of our knowledge, this is the first survey of its kind. We discuss both SDN- and non-SDN-dependent research on network verification. We include both formal methods that rely on well-defined (and commonly abstracted) models of the network and empirical, system-oriented approaches that operate on top of actual network. We investigate approaches that assume either a static network configuration at verification time or operate under change in the network state, including the controller, network devices and packet queues. We believe that a comprehensive and systematic survey of network verification approaches is timely and necessary; research has been extensive in the last decade and involves diverse areas and communities, therefore it is important to consolidate knowledge within a unified taxonomy. This will allow researchers from both networks/systems and software/hardware verification communities to understand the existing literature and, hopefully, enable future transdisciplinary collaborations.

II. Control and Data Plane

In a major move towards softwarisation, *software-defined networking* (SDN) [3] introduces pure decision-making logic abstraction. Whilst, conventionally, packet forwarding and routing take place in the same network box, software-based networks outsource intelligence (routing and other network functions) from the custom ASICs to a domain-specific software application, running on a general-purpose server. The SDN platform consists of three successive layers: an underlying network infrastructure layer (*forwarding/data plane*), a control layer (*control plane*) and an application layer. The control plane[2] provides a single system-wide access interface to programmers and operators. The communication between the control and forwarding layer is achieved through a vendor-agnostic interface which is referred to as southbound API. OpenFlow [6], [59] is the most popular actualisation of data-plane abstraction.

OpenFlow-enabled switches consist of an array of flow tables, a set of ports and an open-source control protocol daemon, i.e. OpenFlow. Flow tables consist of flow entries sorted by priority, and implement two functions: classification and forwarding; The former is based on match conditions (patterns) on a set of incoming packets' header fields. If a match is found, the matched packets, namely flow, are forwarded out the egress interface that the action associated with the highest priority matching entry will return; Else, if the ingress flow does not hit any entry, the fate of the flow relies upon the configuration of the table-miss entry, if such entry exists in the table, which has the lowest priority (0) and defines how to process unmatched packets (drop, pass to another table or send flow to the controller). The OpenFlow switch daemon translates the high level network policies into plain OpenFlow primitives to set up data paths in the data plane, enabling thus the controller to manipulate the flow tables of the switches.

Based on these abstractions, SDN provides network behaviour programmability, automation, homogeneous visibility and standardised representation of network configuration. The traditional network services which are currently deployed as middleboxes, such as firewalls, Network Address Translators, WAN Optimisers, Load balancers, etc, can then be implemented using open APIs on the controller. As an API driven controlling paradigm, SDN adjusts networking to a software-oriented culture, free of distributed control protocols, allowing network administrators to view the network as a whole. This advancement creates an opportunity to reconsider the workflow of network monitoring and debugging.

Even with such benefits in place that SDN might provide, there are equally risks including new challenges for network operations tools to keep pace with a significant amount of state with respect to (1) the topology, i.e., switches/routers, clients, middleboxes, links, flow table entries, queues, packets and their header fields[3], (2) the controller program, and (3) many events related to highly dynamic network state changes (table entries installation/removal, packet arrival, flow entry hits, table-misses, packet injection from controller, controller program changes, etc), and the orderings over them. Moreover, due to the phenomenon of inter-packet interference[4], the number of threads to be explored increases significantly. Similarly, control messages between the controller and the switches may be processed in an arbitrary order and this may lead to potential bugs, such as race conditions. Also, abstraction breaks traditional networking into dynamic components and

[2]Other terms such as SDN-controller, Network Operating System or Network Hypervisor are also used to represent the control plane, all referring to the same concept.

[3]OpenFlow v1.5 [59], for e.g, supports 45 header match fields, whereas v1.0 only 12.

[4]Inter-packet interference refers to the situation in which the combined processing of two packets has not the same effect under different orderings. Processing, for e.g., packet p_1 firstly, could trigger a tree of events which induces different outcome (state) of processing packet p_2 from the outcome if p_2 was processed first. The more the interferences, the more the interleavings of events.

layers that have to work in unison adding greater performance vulnerabilities to the most basic network functions. In addition, SDN ecosystems consist of highly complicated timing behaviour and complicated control tasks (functions) that are challenging to verify. For these reasons, enhanced bug-free and performance guarantees are required for the SDN architecture to be assured with. Methods like testing and simulation have their advantages but none of them is suitable for exhaustive verification in reasonable time frames. As such, the dynamic nature of SDNs extends the domain of traditional service assurance, making traditional approaches obsolete, requiring so verification techniques that follow dynamic approaches. Yet, existing network analysis solutions can not scale and adapt adequately to meet the verification needs of real and synthetic SDN networks.

III. Network Correctness Properties

Provided a model is available, an invariance property is a predicate on a network system's set of states, which specifies, in some temporal logic, the desired behaviour of the system. With regards to IP networks, the requirement properties can be classified into:

- *Functional requirements*. These properties specify the function that a network or a set of network elements should perform, or in other words, what a network is supposed to be able to do. For this reason they are also called *behavioural properties*. They comprise the main subject of interest in the various verification and testing research efforts [60]–[90]. The typical functional requirements include *topology-oriented* specifications. The functional properties of most interest to verification are *reachability/isolation* between/of (all) pairs of nodes/ports, *loop-freedom*, *packet delivery* (no silent packet loss due to blackholes), complete traffic processing *separation* between different tenants, nonexistence of stale ACL rules, etc. There is another sub-category of more sophisticated (richer) invariants related to fine-grained packet processing and fine-grained computation offloading for resource constrained networks. This subclass includes ▸ *black-listing/whitelisting* (requiring special action against or in favour of a traffic class, e.g. rejecting any, or allowing only, connections - packets to a particular IP address), ▸ reachability via *waypoints* (requiring specific traffic to waypoint through a particular or a chain of network elements), ▸ *non-bypassability*[5], ▸ routing with path *length (hops) constraints*, ▸ *special forwarding consistency*[6], etc.
- *Non-functional (non-behavioural) requirements* are all the requirements that place constraints on functional requirements. The key non-functional aspects of network behaviours, include: (1) *performance*: bandwidth (minimum throughput offered), end-to-end latency bounds (delay), maximum jitter[7], error rate, congestion, and other), (2) *reliability*, i.e. the capability of network to operate without failure in a specific time window in terms of consistency, availability, integrity (e.g., packet integrity, routing/flow table integrity), fault tolerance (e.g., mean-time-to-failure), recoverability, responsiveness, etc. (3) *security* requirements define permissible traffic, access lists, authorization, authentication, etc.

Sometimes, it is not enough to know whether something will or will not happen, one rather needs to have a quantitative estimate, for e.g., of the time when (or the probability that) some situation will arise. Ergo, another classification of properties is that into (a) *qualitative* and (b) *quantitative* ones. As depicted in Fig. 1, there is an overlap between the property classifications. The constrains on how the netfwork will behave, that the non-functional requirements place, can be both qualitative and quantitative, whereas the functional properties are always qualitative. Quality-of-Service (QoS) metrics, for instance, stretch out on all the categories of performance, reliability and security guarantees; Quantitative QoS parameters fall under performance sub-category, while qualitative QoS metrics are more subjective in nature and accept only categorical values, and as such are classified as either reliability or security goals. An example of a quantitative QoS could be: "70% of traffic delivered at service level X will experience no more than 100 ms latency", or a reliability requirement in quantitative terms is "the network shall have no more than 30 packet losses/day". The property: "minimise the end-to-end delay for delay-sensitive applications (e.g., online interactive gaming traffic)"[8], is an example of a qualitative QoS. Verifying quantitative properties require reasoning on *time* (for e.g., "what is the worst-case time for delivering a packet?"), *stochastic information* ("what is the probability of an acknowledgement within 5ms?") and *resources availability*, and it remains as one of the great unexplored avenues of research.

So far, there is no general consensus in the network verification community on neither a precise distinction between functional and nonfunctional properties, nor sub-classifying them. Consider, for example, the situation in which we want to block brute force attacks to an SSH daemon running on a server accessible to the outside world. This could be expressed as a security property: "any unauthorized SSH login attempts shall be denied", which would be classified as a non-functional one. But, in order to implement it, it will be needed to block a specific IP address, or range of addresses, and thus the requirement will be further specialised into: "any packet form the offending IP address 1.2.3.4, received via interface eth1 of

[5]Non-bypassability is a basic security property, which means that an enforcement mechanism should not be bypassed (avoided) without being applied, for e.g. "a traffic for a VLAN interface should not bypass the firewall".

[6]An example of forwarding consistency property is the *multipath consistency* that Batfish [88] introduced. This property asserts that, for networks using multipath routing, it should never be the case that a packet is dropped along one path but not the other.

[7]*jitter buffer* in VoIP (Voice over IP) networking, is a shared data area where voice packets can be collected, stored, and sent to the voice processor in evenly spaced intervals. Variations in packet arrival time, called jitter, can occur because of network congestion, timing drift, or route changes.

[8]In a conventional network, this specification can be concretised by computing the best path using, for e.g., EIGRP updates metrics.

the SSH server, should be dropped". In an SDN setting this property would be formulated as: "there should exist a flow entry e in the flow table of the SSH server, which matches packet p with source IP 1.2.3.4 and TCP port 22, and has an empty instruction list[9], and furthermore, if the packet p is matched by multiple flow table entries, e should have the highest priority". Magically, the latter falls now under the functional requirement class by simply expressing the initial security property as a behavioural problem.

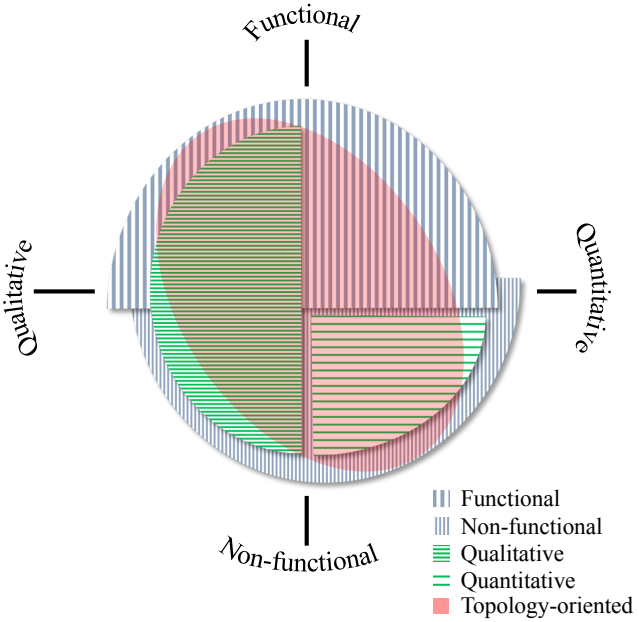


Fig. 1: Pictorial representation of the main types of network correctness properties.

Another differentiation, distinct though, is generally made between *safety* and *liveness* (or *progress*) properties. Informally speaking, safety properties are usually characterised as *"nothing bad will happen"* ($\Box\neg\text{☹}$), in the sense that a given set of (bad) states of the model are not ever visited. Liveness properties can be either (i) *guarantee* or (ii) *response*. A guarantee property asserts that "*something 'good' will eventually happen*", i.e., a requirement of interest, should eventually be fulfilled ($\Diamond\text{☺}$). Another variant of progress requirements is *recurrence* expressed by the canonical formula of the type $\Box\Diamond\text{☺}$, states that something good happens infinitely often. A response formula is $\Box(\varphi \Rightarrow \Diamond\psi)$, which asserts that ψ is guaranteed response to φ. Safety properties are violated (falsified) in finite time, i.e., by finite system runs (finite trace prefixes), whereas liveness properties are violated in infinite time. Other property types are *stability* or *persistence* ($\Diamond\Box\varphi$), *correlation* ($\Diamond\varphi \Rightarrow \Diamond\psi$), *precedence* ($\varphi \Rightarrow \psi_i \,\mathcal{U}\, \psi_j$), *objective* ($\varphi \Rightarrow \psi_i \vee \psi_j$), et cetera.

Another class of property specifications for networks involving timing constraints, is that of real-time requirements. As LTL is a discrete-time logic, a medley range of extensions of it have been proposed for declaring specifications in real-time settings [91]–[93]. *Metric Interval Temporal Logic* (MITL) [91], for example, which is the most renowned one, constraints temporal connectives by intervals. For example, in an SDN, a classic controller-switch interaction requirement for bounded response time that *"every Packet-In message directed to the controller must be followed by a response Packet-Out within 1 time unit"* is expressed by the MITL formula: $\Box(p_{in} \Rightarrow \Diamond_{\leq 1}\ p_{out})$. Timed Propositional Temporal Logic (TPTL) [92] is another timed extension of LTL which employs *clock variables* to quantify statements about time progress. For instance, the above property can be expressed in TPTL by the formula $\Box\big(p_{in} \Rightarrow x.\Diamond(p_{out} \wedge x \leq 1)\big)$, where "$x.\Phi$" means that clock x is reset at the time a `Packet-In` message is sent, before evaluating Φ.

[9]The packet is implicitly dropped if there are no OUTPUT instructions in its `action_set` to be executed.

IV. Taxonomy of Network Verification Approaches

This chapter discusses and explains the criteria applied for comparison of the literature in the following chapter. The criteria are somewhat ordered by the level of importance to quickly grasp the approach in question.

Properties This criterion discusses what kind of network properties are being checked. These can be topological (reachability) or more advanced like blacklisting (see Section III). More detailed restriction of the properties expressible will become clear from the criterion *expressivity* discussed further below.

Model This criterion concerns the model of the network. We distinguish formal and non-formal models. The type of model has huge impact on the techniques available to express and check its properties. Labelled transition systems (and other kind of finite state machines) are typical kinds of formal models. The topology of the data plane of a model is often represented as a graph.

Specification Language This states what language is used to express the network properties. This may be a formal logic, like *linear-time temporal logic* (LTL) using a certain number of basic predicates to express features of the network. It can also be a entirely bespoke language. If the properties are fixed and built-in there may be actually no need for such a language altogether.

Type of Check: This criterion is referring to the type of checks the network properties are subject to. A specification of a property can be used to find bugs violating the property or to verify the invariance of the property during the entire runtime of the network. The latter is of course much stronger and more complicated and costly. In some cases where a language abstraction is suggested with a compilation to a low level network language – in order to improve more reliable programming – there is no explicit property or check, so the type of check then refers to the soundness of the compilation itself.

Checking Phase This criterion addresses the network program phase at which checks are carried out. This can happen

statically (offline) or dynamically (at runtime) and as such examines just the execution paths and variable values invoked during execution, or it is performed offline in a non-runtime environment (static analysis) allowing even for an exhaustive checking. It can also be the case that an approach does a mixture of both.

Layer This criterion details which layer the approach is concerned with: data plane or control plane, or possibly both. For SDN related approaches, of course the control layer must be involved. This information is highlighted but will be implicitly be part of the methodology explained below.

Methodology This aspect concerns the main approach (methodology) employed for the suggested analysis. Of course, this will depend on what model is used and how properties are expressed. It will be explained which established techniques are in use. For instance, model checking or bounded model checking or symbolic execution might be applied. Graph algorithms may be used for reachability analysis. Often, a combination of various different techniques is in use and this needs to be explained.

Expressivity The actual range of network properties that can be analysed or verified with the given approach is discussed. On one end of the spectrum, there is no flexibility whatsoever, and only a few properties are built-in. On the other extreme end, there is a proper language for the user to define a wide range of properties. Usually such a language is based on first-order or temporal logic and uses built-in predicates to specify properties of the network, e.g. the flow table state in a switch. The exact nature of those predicates then again limits the expressivity of network properties.

Experimentation This criterion considers the range of examples that have been dealt with, and can be dealt with in principle, according to the authors of the tool. Where sufficient information is available this may include the size of the examples. Typically, authors provide a survey about their experiments and indicate how large the network is (in terms of number of switches) and what kind of properties have been analysed.

Deployability This criterion relates to the resources of the tool, in particular resource consumption in terms of execution time and memory. It also comprises potential optimisation suggested or implemented. If so, those optimisations are briefly explained and it is stated whether there are proofs to show the soundness of the optimisation. The minimal objective here is to not change the semantics, i.e. the behaviour, of the network, i.e. establish a simulation relation. A simulation relation might potentially lose some of the possible traces, the gold standard is to preserve all of the behaviour. The latter is essential for verification purposes.

Limitations This criterion points out specific restrictions and shortcomings of the approach and tool in question. For instance, state explosion in model checking based approaches is a typical candidate and it may limit the size of the network that can be checked by the tool. The limitations can address the following criteria from above: performance, experimentation, expressivity, timing.

V. Survey of Network Verification Approaches

A. Kuai [72]

Properties. Kuai [72] is concerned with verifying safety properties of SDN networks. These properties concern the correct deployment of all kinds of network policies with the help of the controller program. The deployment usually involves installation and removal of forwarding rules in switches.

Model. The network is modelled finite state transition system based on an interleaving semantics, where concurrency of actions is reduced to the non-deterministic choice among their possible sequentialisations. The finite state transition system then can be analysed using model checking. Some abstractions and simplifications are required to achieve that. The transitions system is modelled as interleaving of several smaller finite transition systems that communicate via queues that are part of the overall network state: each client (transition subsystem) has a packet-in queue and can non-deterministically send a specified packet (to a connected switch) or receive a packet from its queue (removing it from there) and concurrently send packets; each switch (transition subsystem) has a set of ports (for forwarding and receiving packets), a packet in-queue (*pq*), a control queue (*cq*), a forwarding queue (*fq*), a flow table (*ft*) and a *wait* flag for synchronisation. The forwarding queue stores the packets forwarded from the controller and the control queue *cq* receives any control message from the controller. Switches send packets they don't know how to handle to the controller's request queue (*rq*). These components are summarised in Figure 2.

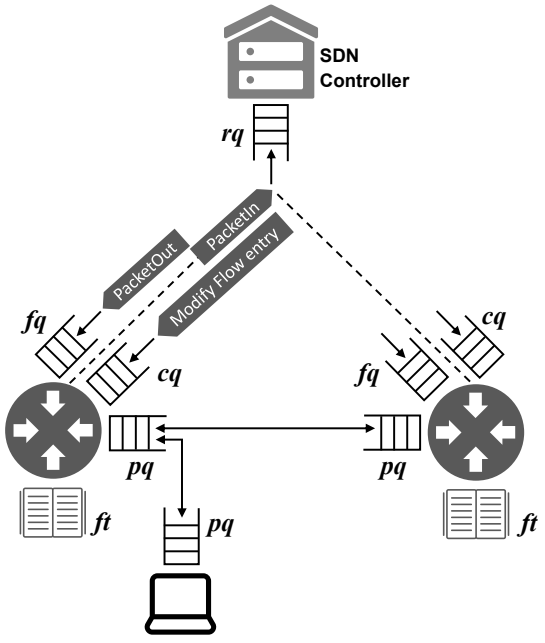


Fig. 2: The queues and connections of the KUAI model

A flow table *ft* is a collection of prioritised forwarding OpenFlow rules; a rule consists of a *Match Set*, an *Action Set* and *Priority*. A control queue *cq* contains control messages (or

barriers) sent from the controller in order to update a switch's flow table, i.e. add, delete or modify flow/group entries in the OpenFlow tables.

A switch can execute a variety of actions (transitions): In its standard mode of operation, it can match a packet available in its packet in queue with a rule and forward it along its ports as described by the rule, or if there is no matching rule for the packets (and there is no barrier message in the control queue for the switch) it can be put in the request queue to the controller (for it to process) and set its wait flag to change into "waiting mode" for the controller to decide how to deal with that packet. If there is a control message in its control queue, and there is no barrier message in its control queue, then it can execute that command, be it a rule installation or deletion. In waiting mode it may also receive a forwarding message from the controller and if there is no barrier in the control queue then the switch forwards the packet accordingly and unsets the wait flag. If there is a barrier message in the control queue the only action for the switch possible is to dequeue all control messages up to the first barrier and update itself accordingly to those messages. A barrier message is a synchronisation mechanism for the controller to force a switch to update itself before it continues processing any forwarding actions.

The controller program is modelled as a transition (sub) system which is basically an automaton. Depending on its state and the packet found in its request queue (into which the switches write), it changes into a new state, removes the packet from the queue and responds by sending two kinds of messages. A forwarding message instructs the switch how to process the packet in question. A pair (*pkt, ports*) will arrive at the switch's forward queue through which the controller orders to forward packet *pkt* along the ports *ports*. Optionally, the controller may also send concurrently any finite number of control messages to the control queue of any subset of switches of the network. The controller, after all, is in charge of maintaining and configuring the network switches.

For the forwarding operations the model uses a function describing the network topology. A packet contains bits describing (an abstraction of) the proof-relevant header-field information only (depending on the example) and its location which is a port in a switch. So for the purpose of the verification one abstracts away as much detail as possible and stores in packets only information relevant to the routing of the packet.

Queues are modelled as multisets with a nondeterministic dequeueing action that implicitly models the random arrival time of packets in the queue. The multiset implementation means queues are bounded. In order to get a finite system, there must be only finitely many multisets. Packets are already finite bit vectors, and also control messages are finite. It remains to have a bound for the possible number of multisets. A packet in a multiset is therefore assumed to appear either not at all or an arbitrary (unbounded) amount of times, i.e. either 0 or ∞ many times. This is called the $(0, \infty)$ abstraction. With this abstraction all queues are finite state (multisets).

Specification Language. Network properties can be expressed in linear temporal logic, with only the box operator available (safety properties only!), with built-in base predicates that are assertion over packet fields and over control states. The state of the switches or any queues can not be reasoned over, they are the "internal state" one observes via the behaviour of packets in the network controlled by the control program. For packets this means that their bits can be inspected including source and destination address and any relevant information like or protocol information stored. The precise form of packet information depends on the example, of course.
Type of Check.: One can find bugs and traces of execution that violate the safety property in question. Proving the invariance of the safety property for all possible execution traces (verification of the absence of any bug) is only possible if all traces are actually checked. It appears that for the examples checked verification has been achieved. However, this verification is done for an optimised transition system
Checking Phase. This approach uses a model of the network and analyses it offline (static). One can do this before switching it on.
Layer. As one can reason over packets and their position in the network, one can reason about the data plane. The possibility of addressing the control state allows one to reason about the controller program and thus the control plane.
Methodology. The methodology in use is model checking. A safety property is checked against all (or a subset) of the traces of the network model. In order to combat the state explosion problem several optimisations are in place (see below).
Expressivity. Due to the fact that a subset of linear temporal logic is in use, a wide range of safety properties can be expressed. Liveness properties cannot be expressed. Change of topology cannot be expressed.
Experimentation. Various examples have been reported. It has been checked whether an SSH controller in a network with 2 switches and 2 clients actually blocks SSH packets from arriving. A MAC learning controller based on a POX implementation of the standard ethernet discovery protocol has been checked for forwarding loops. For this experiment, the packets carry history bit for every switch and store whether they have already been at said switch. A controller implementing a single switch firewall, as well as multiple switch replicated firewall have been tested. The controllers store in their finite state a configuration file to describe the flow between two clients that is allowed. The property checked is that packets are only dropped between clients if the configuration file of the controller does not contain this pair. A controller implementing the Resonance [94] algorithm to ensure security in large networks has been analysed to ensure that packets from so-called quarantined states cannot be forwarded. An policy enforcement layer built on top of OpenFlow, 'Simple' [95], has been experimented with as well.
Deployability. In order to deal with the examples as mentioned above, for even small networks one needs optimisations due to the well known state space explosion problem. Optimisations include restricting the request queue of the controller to size

one, restricting switches to execute no-match events only if the request queue is not full, merging control actions and immediately succeeding barrier actions, etc. Optimisations, which are stutter bisimulation equivalences, are proved sound. Two of the reductions, however, the $(0, \infty)$ and 'all Packets in One Shot', are simulations (stutter trace inclusion) which means that new behaviour may be added in the abstract transition system. Most examples above are run with a small number of clients and switches, usually around 5 to 10. Without optimisations only the verification of the tiniest network terminates in reasonable time. For the slightly larger networks the runtime ranges from a few seconds to a few hundred seconds. The largest number of states visited was almost 24 million. Runtime is proportional to the number of states visited.

Kuai is implemented on top of PReach [96], a distributed enumerative model checker itself built on Murphi [97]. The different transitions subsystems were modelled as different implementation used 4TB of RAM and 150 cores.

Limitations. The size of networks that can be checked can't be too large (about 10 switches). Only safety properties can be verified. The network semantics assumes that some actions concur synchronously that in reality can occur concurrently. Barriers are always executed first . The topology is fixed.

B. Header Space Analysis

Properties Three main categories of functional, topology-oriented properties are checked in the Header Space Analysis (HSA) work [60]: reachability between hosts, lack of forwarding loops and isolation of network slices. Reachability is generalised to check also other path predicates such as: black hole freedom, routing via a waypoint, maximum hop count (length of path never exceeds a threshold), isolation of paths (for e.g., http and https traffic do not share the same path), etc.

Model The framework used in this approach is built on a geometric model. Packet headers and flows are represented geometrically in a Boolean space of header bits. An L-bit packet header is modelled as a bit-vector, i.e. a point in the geometric space $\{0,1\}^L$, where each bit of the header corresponds to one dimension of this space. Payload is abstracted away from the packet. Flows are modelled as regions in this space, representing all the packets in the flow. Network boxes are modelled using a *Switch Transfer Function T*, which transforms a header h received on ingress port p to a set of packet headers on one or more egress ports: $T : (h, p) \rightarrow \{(h_1, p_1), (h_2, p_2), ...\}$. The *Network Space* is the space of all ingress headers in the network. Each transfer function is given by an ordered set of rules. A rule typically consists of a set of physical input ports, a match condition (pattern-proposition), and a set of actions to be performed on matched packets. Actions can forward out to an interface, discard, modify the values of specific header fields (rewriting, encapsulation, decapsulation). In this sense, boxes are abstracted as set of conditional expressions. The overall behaviour of the network is represented as a piecewise network transfer function combining all the switch transfer functions. The network topology is modelled using a *Topology Transfer Function*. For instance, if port p_{src} is connected to p_{dst} using a link, then this function maps (h, p_{src}) to (h, p_{dst}), modelling the fact the header h is transferred from p_{src} to p_{dst}. The conventional, two-valued Boolean domain $\{0,1\}$ is extended into a ternary one by the addition of a special, "unknown" third value, denoted by '$*$'. This is a wildcard character which is treated as (matching) either a "1" or a "0". As a result, the regions in the header space (hypercubes) are represented as sequences (referred to as wildcard expressions) whose domain is the ternary one. Input test packets may be parametrised symbolically by Boolean variables (i.e., symbolic simulation). Combining ternary modelling with symbolic simulation, and injecting fully wildcarded test packets, enables the exploration of the entire state space. The overall model can be thought of as a propagation graph where each vertex represents a tuple of a packet header set and an ingress port this set has reached to, and each outgoing edge is labelled with the transfer function of the box the ingress port belongs to. To work out the header spaces left on each hop, a set algebra is introduced.

While the headers may have been transformed in the packets journey, the original headers sent by the sender can be recovered by applying the *inverse transfer function*. For example, in Figure 3 an $all - x$ (wildcard) test packet header is injected into port 1 of router's miniature model R (i.e., port R_1). To keep things simple, 3-bit headers are used in this simplistic scenario. The router R transforms the $all - x$ header space as a result of flow table rules (Forward & Rewrite: rewrite bits `1xx` with value `0xx`) that are filtering out some space of it. Then, it can trivially be traced the remained header spaces backwards (using the range inverse) to find the set of packets host-A can send to reach host-B.

Specification Language Properties are specified implicitly as code snippets in a library of tools written in Python, called Hassel. Algorithmically, they all fall into the category of computing reachability sets of packets.

Type of Check Header space analysis provides the full set of failed packet headers as counterexample. However, with regard to the exhaustiveness, in the case of loop detection, for example, the authors claim that they detect all loops by injecting an $all - *$ test packet header and tracking the packet until it returns to the port it was injected from.

Checking Phase It is a static checker which can be applied to snapshots of the network only.

Layer HSA reasons purely on forwarding state.

Methodology As a custom design with its data structures and algorithms, it computes a reachability tree, i.e., all the paths along which a nonempty header space is left at each vertex. By modelling packets as points in an L-dimensional space, two abstractions are achieved: (1) all protocols/layers are collapsed into a flat (protocol-agnostic) sequence of bits, and (2) header bits irrelevant to forwarding are ignored by the use of wildcards. Three simulation-based algorithms (one per each main property: reachability, loop detection, slice isolation) are used to analyse the reaction of the network on injected test packets, using symbolic representation of their

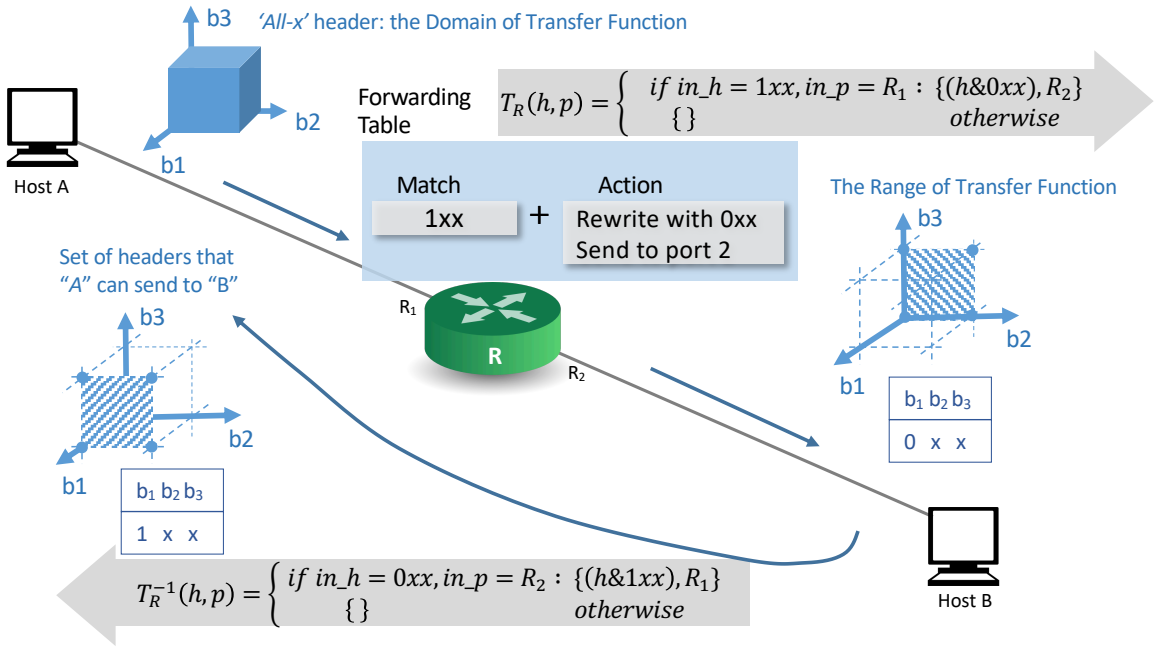


Fig. 3: The *Inverse Transfer Function* (T_R^{-1}) is helpful for detecting reachability failures and loops which require tracing backwards from a range (`0xx`) to determine what header set host-A can send to B (`1xx`).

header fields, instead of testing each concrete example.

Expressivity This framework supports three main categories of hard coded invariant policies, disallowing the verification of more expressive network properties that a specification language would offer.

Experimentation The checking performance for all three properties was measured in a relatively large production network (Stanford Network). In the loop detection experiment, test packets were injecting from 30 ports. Slicing, which is a generalisation of VLANs, is another experimentation subject regarding spaces overlapping and packet leakages. A network slice is a logical sub-network which runs on top of a shared physical network infrastructure combining resource virtualisation with the isolation level demanded. More formally, a slice is a tuple of network boxes, ports, topology function and a set of predicates on packet headers. In order to check whether two slices do not overlap, the intersection of their header spaces on every port of the slice is computed. The reachability experiment is run by injecting a symbolic packet from a router, and as the transfer function rules hack away at input hypercubes along the path, the left space (if there is any left) seen at the destination is the range of the reachability function.

Deployability To gain algorithmic leverage, the domain structure exploited incorporates small equivalence classes by treating groups of headers as an equivalence class wherever possible (for e.g., the union $110* \cup 100*$ simplifies to $1*0*$). Another key algorithmic optimisation is the compression via the difference of hypercubes (lazy subtraction). This optimisation consists in augmenting the notion of header space objects by allowing them to be represented as a subtraction of unions rather than as just a union of wildcard expressions. Then, there is no need the computations of header set subtractions to be performed actively in a stepwise manner but can lazily be postponed until the end of the path. Other algorithmic optimisations used are lazy evaluation, dead space elimination, and IP table compression. All optimisations are orthogonal. To maximize performance, on top of the above, some memory optimization and parallelism techniques are deployed in the C version of the algorithms. The python version needs 560 seconds to run the loop detection test for all 30 ports, where 12 loops were found, while the C version only takes 2 seconds. However, it takes 151 seconds to compress the forwarding table and to generate the transfer functions on a preliminary stage. Concerning the reachability test, the verification running time is $O(dR^2)$, where d is the the maximum number of hops needed for a packet to reach the destination, and R is the maximum number of rules in a box along the path. In numbers, the run time of 13 sec is reported for the algorithm to compute the reachability from a router to another one in the Stanford backbone network. The time for checking the slice isolation is quadratic in the number of wildcard expressions per slice and linear in the number of slices.

Limitations In a static manner, HSA extracts the forwarding rules from dumps of the switches' routing tables to analyse them, and as such it lacks the capability of having a consistent view of the data plane's behaviour at small time scales. Although the entire space is exercised by pushing totally

wildcarded headers through every port, each packet is tracked until the first loop is found, and the question that arises is what if there exist multiple loops through which a packet might turn back to the same injecting port. That might leave deeper potential loops in the execution paths unexplored. The algorithms don't seem to generate iteratively multiple counterexamples per injection port, and in this sense the exploration seems not to be exhaustive. With respect to the specification language, a more abstract formal specification language would allow to check the network behavioural requirements at a higher level, rather than directly probing into the code. Than, it would be enough to prove whether the low-level code is a refinement of the specification. Last but not least, HSA does not model stateful functions.[10]

C. VeriCon [74]

Properties. VeriCon [74] is concerned with statically verifying safety properties of SDN networks. The controllers are written in a proposed language called CSDN (C for core). The properties concern the correct deployment of all kinds of network protocols with the help of the controller program. The deployment usually involves installation and removal of forwarding rules in switches.

Model. The network is modelled as a set of relations. VeriCon receives three inputs, the SDN controller program, a first-order formula describing constraints on the network topology, and the safety property to be shown for the network. There are predefined relations describing direct links or paths between ports of switches or switch port and a host, respectively. These formulate constraints on the network topology. VeriCon is quite permissive in the sense that it verifies w.r.t an arbitrary network topology that meets the given topology constraint, also called topology invariant, rather than a fixed topology. In other words, topology changes at runtime are allowed as long as they satisfy the topology constraint. Typical invariants of a topology may include absence of self-loops or the fact that packets can only arrive from reachable hosts.

Packet (headers) are modelled as pairs of host source and destination address. Additional header fields, when needed, are modelled as functions on packets. Further built-in relations involving packets describe a switch's flow table, the arrival of a packet at a switch's ingress port, and the fact that a packet at a switch has been forwarded from an ingress to an egress port, the so-called "history relation". Forwarding events are recorded in this history relation. Since rules are modelled as tuples in a relation (describing a flow table) they can contain information like the rule's priority.

The controller is modelled as an infinite loop consisting of guarded commands manipulating the relations. The guards model switch and controller events like packet forwarding at switches or incoming packets at the controller. Note that for the forwarding event the controller's command is fixed in the sense that the packet must be forwarded according the rules in the flow table of the switch in question. As there is no explicit switch semantics, the behaviour of switches is implicitly modelled by the controller in that way. The user-defined commands of the controller program can only be defined as reaction to packet-in events, i.e. when the controller receives a packet.

The command language is a simple imperative block-structured language with assignment, sequential composition, while loop and conditional. It includes commands for adding a set of tuples to, and removing a set of tuples from a relation. This, the programming language provides relations as user-definable data type. One can simply view these relations as tables of a relational database. These tables will often contain host addresses, switch names and port numbers, such that the controller program can memorise relevant facts. Since flow tables are modelled as relations, this allows the controller program to install or deinstall a rule at a switch. Forwarding a packet is modelled by inserting a tuple into the special "sent" relation described above. Flooding a packet to all switch ports except the packet's ingress port is another possible command. The guards in the conditional and while-loop use boolean expressions and the check whether a tuple is in a relation is such a possible expression.

The controller program allows initialisation of user-defined relations before the event handling loop. Semantically, a command is a predicate transformer, which given a predicate (involving relations describing the controller and network) specifying the post-condition after execution yields the weakest liberal precondition that needs to hold to guarantee that the predicate holds as postcondition. The controller comes with a user-defined transition invariant that specifies the intended use of the data (i.e. the user defined relations) manipulated by the controller.

It is assumed that switch and controller events are executed atomically.

Specification Language. The invariants, the topology, safety and transition (controller) invariant are all expressed in first-order logic.

Type of Check.: The method will prove that the given safety property (invariant) holds in whatever order the networks events are processed. In case the property does not halt, bugs that violate it can be reported.

Checking Phase. This approach uses a model of the network and analyses it offline (static).

Layer One can reason about the data plane via the built-in predicates. Since the controller program is the one that is formally verified, one can also reason about the control plane.

Methodology. Hoare-style reasoning [98] is used to generate verifications conditions that are handed over to the SAT solver. Let Inv be the inductive invariant, that is the conjunction of topology, safety and transition invariants. The weakest liberal precondition $wp\,[\![event \Rightarrow cmd]\!]$ is computed for every event/command pair in the controller program. If $Inv \wedge \neg wp\,[\![event \Rightarrow cmd]\!]\,(Inv)$ can be shown to hold by the SAT solver for any event then the invariant is not preserved by this event and a bug has been discovered. Otherwise, the controller program is proven to preserve the invariant. First, it

[10] A subset of network functions that require to keep algorithmic state.

is checked that the invariants are consistent in the initial state. Moreover, the inductive invariant is automatically obtained by the initial invariant by iteratively apply ing the weakest precondition operator until one actually obtains an invariant for the controller program events. Note that while loops need to be annotated with a user-defined invariant to start with.

Expressivity Due to the fact that inductive invariants are proven, only safety properties can be shown. Those range over first-order logic expressions that can use the relations used to express network and control state.

Experimentation Various examples have been reported: a simple stateful and well as stateless firewall, a firewall that allows for migration of trusted hosts, network authentication with learning as in Resonance [94], Stratos [99] style traffic steering, called middlebox composition.

Deployability VeriCon is implemented in Python and uses the Z3 [100] SAT solver. Examples run reasonably fast, under 0.3 seconds, despite several thousand verification conditions may be generated for the SAT solver to check. However, the total number of controller program lines never exceeded 93.

Limitations This approach can only verify safety properties. All events are assumed to be executed atomically, so bugs due to race conditions, i.e. installations out of intended order, cannot be detected. A potential limitation is the power of the SAT solver to check the verification conditions as they are first-order in general and contain quantifier nesting like $\forall\exists$. The authors argue that "by observation" the instantiation dependencies for existential quantified variables are "shallow", i.e. they do not create new instantiation opportunities. It remains rather vague, however, as this is just an observation based on a few experiments.

D. NetPlumber

Properties NetPlumber [63] builds on HSA [60], deriving its property set therefrom.

Model NetPlumber is centred around the idea of modelling headers as points and packet flows as regions in the so-called header space [60] ($\{0,1\}^L$ where L is the length of the headers). Its internal model is graph-based in which nodes represent OpenFlow-like forwarding rules in the network (drawn from the FIBs (Forwarding Information Base)), and directed edges of the graph represent the next-hop dependencies of the rules. A rule is said to have a next hop dependency to another rule if, on the premise that there exists a physical link connecting the rules' switches, the range of the sender-switch's transfer function intersects (meets) the domain of the receiver. This intersection represents a possible flow path, and for this reason the edges are also referred to as pipes, the intersections as pipe filters, and, by extension, the graph as plumbing graph. The pipe filter: 111xx010, for example, which is the intersection of the range 111xxxxx of rule r_1 with the domain xxxxx010 of rule r_2, represents the packet headers at the output of rule r_1 that r_2 matches. For pushing flow into the dependency graph, in addition to the rule-nodes, the so-called source-nodes are connected to the graph, and their only role is to generate flows. The generated flows are absorbed by sink-nodes. Another type of node is the probe node. A probe-node is used to monitor flows received on a set of ports, according to the policy at hand, evaluating the constraints on flows. These constraints are of two types: the filter expression, which matches the flows, and the test constraint, which states conditions imposed on the matching flows.

Taking full advantage of Software Defined Network (SDN), an agent sits in-line with the SDN controller tapping into the communication between the controller and switches. The plumbing graph state is updated dynamically at each event occurring in the data plane, such as install/remove flow entries and link up/down. This way, only the dependency sub-graph affected by an update has to be traversed.

The flows are augmented by history pointers which correspond to the rules that have processed this flow.

Specification Language A language of regular expressions, called *FlowExp*, is introduced to express conditions on the path and the header of flows. It is designed to check constraints on the history of flows received by probe nodes. Predicates on the path taken by a flow (the test constraint on the history of flows received by probe nodes), and on the header of flows received on a probe node (the filter) can be either existentially or universally quantified. The base predicates can express the shape of paths and headers and can be composed via the standard logical connectives.

For example, the fact that there exists a flow f that satisfies both a filter and test constraint is expressed in *FlowExp* as: $\exists\{f \mid filter(f)\} : test(f)$, and the probe-node which is configured with it will fire if there is no flow f that satisfies the test flow expression.

Type of Check: Checking is performed by exhaustive and incremental flow analysis using the dependency graph representation of the forwarding plane. Checking iteratively the graph for acyclicity (or computing all subgraphs that have at least one cycle in them) remains one of the issues that has not been dealt with.

Checking Phase NetPlumber is verifying the compliance of a stream of network state changes in real time.

Layer Data plane state changes (rules installation/removal, link up/down events) are observed, and any new stream of updates is applied on the dependency graph.

Methodology A graph-based flow analysis is used. Reachability is computed by injecting a header space region, representing a wildcarded test flow, from the source port, propagating it along the edges of the plumbing graph, and computing the subset of flows that reaches the destination rule-node. To guarantee loop freedom, each rule-node inspects the flow history. Black holes are discovered when an ingress-flow on a node which represents a rule with a non-empty set of output ports, won't egress this node.

Expressivity Flowexp offers a more flexible way than HSA to express and check complex policy queries without having to write ad-hoc code for each case. However, in order to specify higher level policies, the policy constructs proposed in the Flow-based Management Language (FML) [101] are used. FML is a declarative policy language for specifying network-

wide connectivity policies about flows, allowing administrators to focus on policy decisions rather than on implementation details.

Experimentation NetPlumber is evaluated on three production networks: Google WAN (52 switches, about 143K OpenFlow rules), Stanford University Backbone Network and Internet2 nationwide backbone network. Checking all-pair connectivity policy on Google WAN, 60% of rule updates can be verified in less than 1ms, while it would take Hassel 100s, at least. By increasing the number of instances of Google WAN, the runtime gets better, however, 5 is the optimum number of instances. The per add-rule run time of NetPlumber for all networks is well under 1ms, while add-link run time takes a few seconds.

Deployability By running HSA [60] snapshot-based computations in an incremental fashion, instead of building the entire graph, allows NetPlumber to outperform HSA. The dependency graph created consists, in the worst case, of R^2 edges where R is the number of rules in the network. Still, dealing with large amount of information about rules and flows in the plumbing graph, involves extensive memory access. To address this issue and scale up to large data planes, the dependency graph is partitioned into clusters forming a distributed policy checker. By parallelising instances of NetPlumber, a reduction in inter-cluster dependencies is achieved.

Limitations As a snapshot-based approach, NetPlumber, like HSA, is not capable of checking state-dependent policies over stateful settings. Another drawback is the high run time for verifying link updates. While network policy violations are detected, NetPlumber cannot provide an automatic and effective violation resolution.

E. NICE [71]

Properties Violations of (network-wide) correctness properties, both safety and liveness, due to bugs in the controller programs are sought to be discovered. A library of common properties to be checked is provided (such as no forwarding loops or no black holes). Optionally, NICE allows programmers to write application-specific correctness properties in Python, both safety and liveness, as assertions over the global network state.

Model NICE models networks as state machines. The system state consists of three components: the states of the controller programs, switches and hosts. Since the *OpenFlow* protocol follows the event-driven programming paradigm, the controller program manipulates the switches' configuration state. In response to events (e.g. packet-in) it executes appropriate callback routines (i.e. event handlers). Each event handler listens to inputs from the underlying network. When an event occurs, the appropriate event-handling code is executed, re-evaluating the corresponding global variables, which gives rise to a new state. The controller program is accordingly modelled as a transition system on (pattern of) events, event handlers and states. The switch state is modelled abstractly as a tuple of communication channels and a flow table. A communication channel is a FIFO buffer and can be of one of two types: packet channel and OpenFlow channel. Transitions can likewise be of two types: `process_pkt` and `process_of`. The former is used for processing data packets (e.g. match or forward a packet), and the latter for OpenFlow messages (e.g. flow-mod). The transitions are enabled once an occurrence (packet or OpenFlow command) appears in the respective channel.

To eliminate repeated computations, the flow table is modelled such that its entries have a unique representation by considering only one ordering. Hosts are classified into clients and servers. The default abstractions of both have two transitions: `send/receive` for the client and `receive/send_reply` for the server. A counter tracks the number of sent packets in the client. The mobile host is a more refined version of the default model the state of which is augmented with location data $(switch, port)$. The location is updated upon firing the transition called `move`. As an open source framework, other models for the hosts are allowed to be programmed.

Specification Language A specific symbolic logic for specifying the properties is not expounded on. The correctness properties are specified in a general unrestricted logic as Python code snippets. So anything that can be programmed is expressible. However, network-wide properties, safety and liveness, require variable quantification and temporal reasoning, so one can probably say that the implemented logic is a first-order temporal one.

Type of Check It is a model-based approach focusing on testing rather than verification. It consists of adapting model checking into a form of systematic testing for finding bugs.

Checking Phase Offline error checking is performed without reference to the controller code runtime behaviour in real-time.

Layer NICE is designed specifically for OpenFlow controller programs written for the NOX-platform.

Methodology Since event handlers react to data plane events, model checking data-dependent apps is tricky; the packet space is huge and enumerating all possible concrete inputs is intractable. NICE deals with this issue by using symbolic execution as a systematic code analysis technique on top of model-checking to identify representative inputs (i.e., equivalence classes of packets) that exercise code paths in the handler. The idea is to execute the event handler symbolically, i.e., with symbolic packets as its argument. A symbolic user input packet is a logical entity, whose header fields can take any possible value. Along each path, when a branch that depends on symbolic input is found, a first-order Boolean symbolic formula (path constraint) that describes the conditions satisfied by this branch is updated. Hence, the path constraints abstractly represent all inputs that induces the code execution to cover the path. The set of all collected symbolic constraints that gives the path conditions is representative of one class of packets. Upon completion of the symbolic execution, a constraint solver is used to solve the constraints and if a satisfying assignment is found, for each identified class, one concrete representative packet is extracted which is to be injected into the event handler covering that path. This way, by concretising the symbolic input, test cases are generated

for covering the path. The controller program under test, will then be executed using the concrete input values (Figure 4).

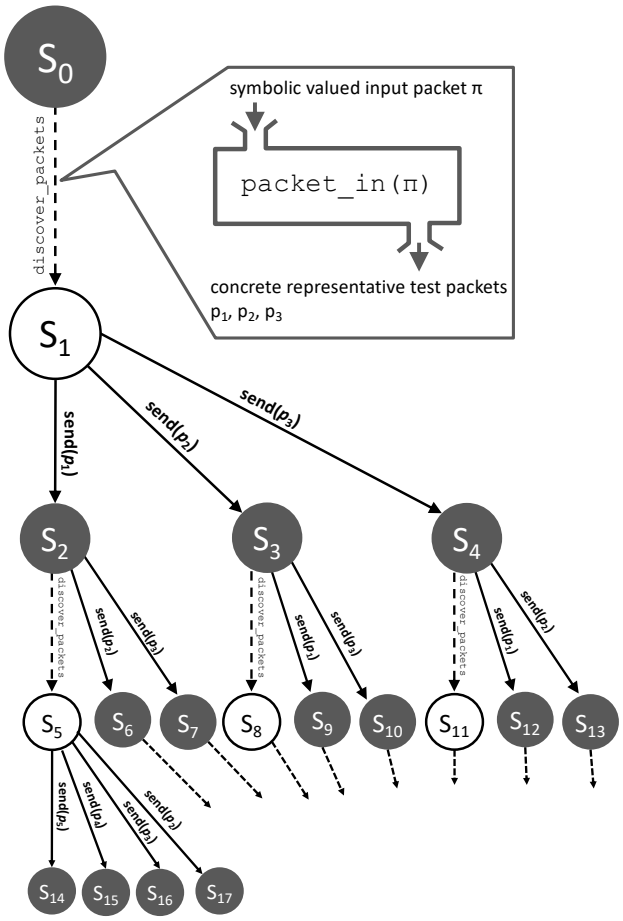


Fig. 4: Example of a concolic execution tree fragment in NICE for one client: Filled circles represent concrete controller states. The `discover_packets` calls the `packet_in` handler with symbolic input packet as argument. Each symbolic state (unfilled circles), obtained as a result of taking the symbolic transition `discover_packets`, represents a tuple of the symbolic store, and the path constraints. The symbolic store associates program variables with expressions over concrete values. On branches with more than one feasible resolution, the symbolic state (S_1 , for e.g.) is forked and all feasible resolutions (three in our example) are explored switching the graph to three new concrete states S_2, S_3 and S_4.

Expressivity Properties are specified using a general-purpose and highly expressive programming language, Python, which hardly can be challenged by any symbolic logic with respect to expressiveness.

Experimentation NICE approach is firstly evaluated in a two-switches topology, each hosting a client, and a MAC-learning switch program in the controller. The property to be checked is not mentioned for this case, however it seems that a property that will allow the transition system to be fully unfolded is considered. A client sends an ICMP echo request packet (ping) to the other client, and the learning switch logic comes into play learning and updating the MAC table with the locations (i.e., the switch and input switch-port) of the senders. The setting is scaled up by increasing the number of concurrent pings. Results show that by employing the canonical representation of the flow tables on top of model checking (without symbolic execution), the state space growth rate slows down two times. In addition, if heuristics are turned on as well, it results in a 28-fold state space reduction for three pings.

In comparison to the other off-the-shelf model checkers, NICE achieves credible results with SPIN [102] performing more efficiently while Java PathFinder (JPF) [103] five time slower. Further, NICE tested three applications: a MAC-learning switch program, a server load-balancer, and energy-aware traffic engineering. While the first application is run in the same two switch/client pairs topology, the load balancer is tested with one client and two servers connected to a single switch, and the energy-efficient traffic engineering app in a network topology with three switches in a triangle, one sender host at one switch and two receivers at another switch. Eleven bugs are reported to have been found in these applications.

Deployability Combinatorial explosion is the main hindrance to application of model checking. Even symbolic execution suffers from limited scalability, particularly with regard to constraint solving cost and path explosion [104]. As a common strategy to ensure feasibility of symbolic execution, NICE introduces search heuristics to prioritise path exploration. The PKT-SEQ heuristic introduces a schedule-driven concurrency control into nondeterministic interleaving of enabled send-transitions. This is achieved by imposing constraints on the outstanding-packets buffer size, resulting in adjusted interleaved executions and, consequentially, in state space reduction. NO-DELAY is another heuristic which, again, reduces the number of thread interleavings of the controller program. This time, the switch-controller communication actions are excluded in the order in which non-deterministic choices are made by the search algorithm (i.e. the total order). The UNUSUAL heuristic elaborates on the fact that the sequential execution (non-interleaved) of switch-controller communication actions, from NO-DELAY heuristic, may hide bugs (e.g. race conditions). For this reason, NICE explores other orderings between those actions in case some unusual and unexpected delay is observed during the execution of the initial sequential ordering. NICE also applies a reduction technique (FLOW-IR) that try to exploit independencies between concurrent packet processing actions, i.e. actions the effect of which is independent of their ordering. This is also known as the commutativity of concurrent transitions feature. The aim is to reduce the number of possible orderings that need to be considered.

Limitations The framework is limited to testing controller applications in Python whose source code is accessible. It

often manoeuvres the model checker in order to control orderings and reduce non-determinism, which might cause loss of observable behaviours. However, in a testing context where any form of guarantee cannot be offered, such manipulations are well suited. Also, a high-level symbolic and non-procedural language to abstract logic and specify correctness properties is not included. Such spec lang would be simpler, smaller, faster and easier to write good compilers for it.

F. Veriflow [61]

Properties. Veriflow can check network-wide, topological properties. such as reachability, loop-freeness, absence of black holes, consistent routing and correctness of access control policies. More specifically, Veriflow exports a custom C++ API that allows network programmers to reason about the behaviour of the data plane; i.e. how packets are forwarded in the network at any time.

Model Veriflow does not directly model the network. Instead, the *forwarding state* is modelled using forwarding graphs. Packets are modelled as *Equivalence Classes (ECs)*; packets belonging to the same EC get identical forwarding behaviour from the network. A forwarding graph is EC-specific; each vertex in the graph represents an EC at the respective (modelled) network device. Edges between said vertices represent a forwarding decision for the EC; i.e. any packet belonging to this EC will be forwarded to the device modelled by the destination vertex in the forwarding graph.

Specification Language. Veriflow exports a C++ API that is used to implicitly express network properties and check respective invariants. The exported API exposes ECs and respective forwarding graphs as well as the effect of a new rule to existing ECs. Verifying a network-wide property for one or more ECs is done by traversing the forwarding graph using an exported C++ method.

Type of Check. Veriflow operates checks on incoming forwarding rules by the SDN controller and is agnostic to the control plane and the respective controller program. For each incoming rule, Veriflow will only calculate forwarding graphs for the affected ECs. For each one of these, it will execute *invariant modules* (i.e. code that implicitly defines network properties as described above). If it is found that the new rule would result in the violation of one or more invariants, an alarm is raised to the network operator.

Checking Phase. Veriflow checks network-wide invariants at runtime by intercepting forwarding rules before they reach their destination network switches.

Layer. Veriflow is a data-plane verification tool that is completely agnostic to the SDN controller and running program. Instead of verifying a network invariant for the whole network state every time a new rule is added, it incrementally verifies invariants by only examining affected ECs and their respective forwarding graphs. When a network property is violated, Veriflow can only raise an alarm for the problematic new rule; this violation cannot be linked to the underlying SDN controller program.

Methodology. Veriflow calculates a forwarding graph for each EC. Incoming rules (from the SDN controller intercepted by Veriflow) may trigger the updating of the set of ECs. A new rule may result in adding or deleting an EC or splitting one to multiple ECs. The respective forwarding graph is then calculated for all updated ECs and invariants are checked by executing invariant modules (see above) that implicitly define the network-wide correctness properties. The *GetAffectedEquivalenceClasses()* method returns the set of ECs that are affected by the incoming rule. *GetForwardingGraph()* returns the forwarding graph for a specific EC. With these two methods one can write C++ programs that can examine all forwarding graphs for all affected ECs. By calling the *ProcessCurrentHop()* method on a specific forwarding graph, a program can traverse the graph, examining the forwarding behaviour for the respective EC and identifying invariant violations. Veriflow uses *tries*, ordered trees that store an associative array, to efficiently store new network rules, find overlapping rules, and compute affected ECs.

Expressivity. Veriflow supports arbitrary C++ programs (called invariant modules) that are executed on specific ECs and forwarding graphs to check the correctness of network-wide properties. Apart from commonly cited reachability properties, the authors demonstrate how Veriflow can be used to check for *conflict detection* and *k-monitoring*; i.e. whether an incoming rule violates isolation of flows between network slices, and ensuring that all flows in the network traverse one of many specific monitoring points, respectively.

The expressivity is constrained by the underlying data that is stored with the forwarding graphs and ECs. For example, the authors acknowledge that Veriflow cannot check performance properties that require knowledge on buffer sizes nor properties that are not implementable in an incremental fashion with respect to only considering affected ECs.

Experimentation. Veriflow's key performance indicator is the verification latency; i.e. the time it takes to verify that an incoming rule would not result in the violation of (implicitly) defined network properties. The authors microbenchmark Veriflow using a stream of rules coming from a simulated Rocketfuel [105] topology and BGP traces. BGP traces were replayed and the resulting updates triggered respective rule generation within the autonomous system (AS). Given the nature of the simulated updates, only the destination IP address is involved in all rules, therefore only this one field contributes to the generation of ECs. Veriflow verified most of the updates within 1ms which is acceptable for real-world deployments. As expected, the verification time heavily depends on the number of ECs that are affected by the incoming rule. For the described setup, the largest verification time was 159.2ms due to an update affecting 511 ECs, although only 5.5% of the rules affected more than one EC. The authors also experimented with link failures that inevitably affect a large number of ECs. In the presence of more fields that affect ECs, Veriflow is unsurprisingly slower; more fields to classify packets translates to more unique ECs that are generated upon inserting new rules. The tested topology consisted of 172 routers. It is not

specified how large the resulting forwarding graphs were, and there is no discussion on how the size of these graphs would affect the verification latency. Veriflow verification is done on forwarding graphs, therefore parallelisation is possible. The effect of Veriflow on user-perceived performance is also evaluated using an emulated network with Mininet [106]. More specifically, the authors measured the overhead Veriflow induces in TCP connections and found that the average time to establish a TCP connection increases if respective rules must be checked by Veriflow.

Deployability. Veriflow is directly deployable in an OpenFlow-based SDN network and operates on the live network. Conceptually, it sits between the OpenFlow controller and the network. It can operate as a proxy for OpenFlow rules or be directly integrated with the controller (NOX [107] in the version presented in the paper). Veriflow intercepts all OpenFlow messages that are sent from the controller to the network and checks whether the insertion of a rule would violate the pre-specified network properties. Veriflow's trie structure is optimised so that header fields that can only have exact values (no wildcards are allowed) are represented in a single trie dimension. Problematic rules raise respective alarms to the network operator.

Limitations. Veriflow will identify rules that, when applied, would result in the violation of user-defined, network-wide properties, however, it cannot link the problematic rule with the running controller program. It is up to the network operator/programmer to identify the root cause of the problem (e.g. a bug in the controller program). Veriflow's verification latency is evaluated with respect to the number of ECs that are affected by an incoming rule, but it is unclear how the size of the network and respective forwarding graphs would affect Veriflow's performance. Moreover, although the number of affected ECs in the presented evaluation scenarios is usually very small, it is unclear how Veriflow would perform in large-scale realistic scenarios with complex network applications. Finally, in many cases, an update would require network-wide changes that would be carried in a sequence of rules destined to different switches. By looking at a single such rule, a topological invariant (e.g. reachability) would be violated; only when all these rules are established, the invariant would hold again. Veriflow does not appear to support such bulk updates.

G. Anteater [62]

Properties. The main invariant properties that can be checked are reachability, loop-freedom, black holes freedom and consistency of forwarding rules between routers. Reachability algorithm serves as a basis for checking the other properties.

Model. The network is modelled as a directed graph with vertices corresponding to network boxes or destinations in the network, and edges representing connections between vertices. Another component of the graph is the policy function $\mathcal{P}$, defined on the edges. This function, which is encoded as a boolean formula over a symbolic packet exercising the edge in question, can express different policies, like forwarding, packet filtering, and transformations of the packet. A packet can be forwarded/blocked over an edge only if the overall policy function over this edge is evaluated to true/false. For e.g., $\mathcal{P}(sw_i, sw_j) = dest\_ip \in 1.2.3.4/16 \wedge tcp\_dest\_port \neq 22$ is a packet filtering rule which blocks SSH access to the IP range 1.2.3.4/16 for packets flowing from switch i to j.

In order to model packet rewrite, each packet is represented by an array of its lifespan instances, where each element of this array represents the state of the packet at each transformation hop. By preserving the history of the packet, each transformation is expressed as a constraint on its history, rather than transforming the same original packet. The transformation constraints are, of course, considered on top of the policy constraints.

Specification Language. Anteater enables access to its objects from Ruby and SLang, which allows properties to be expressed sufficiently via either Ruby scripts or SLang queries.

Type of Check. Anteater verifies whether the network satisfies the property: in case a bug is found, a counterexample is returned.

Checking Phase. The diagnosing approach works offline, and is based on static analysis of built data plane snapshots.

Layer. Anteater checks invariants exclusively in the data plane.

Methodology. In order to capture the data plane state, Anteater collects, via SNMP, the devices' forwarding information bases (FIBs). Then, it combines the invariant and the network description encoding them into instances of boolean satisfiability problem (SAT), and resolves them by passing into an off-the-shelf SAT solver. Reachability, which is the most trivial invariant property here, is checked in a quite classic way: node j (network box or specific destination) is reachable from i, if there exists a packet and a firing sequence $i \rightsquigarrow j$ of edges in the graph such that all the constraints of the policy function along it hold for this packet. For checking whether there are forwarding loops, the graph is rebuilt by creating clone vertices, each of which has the same set of incoming edges (and policies) as the original. Thus, a forwarding loop equate to a clone being reachable from its original. The graph is examined for black holes towards a set of destinations, by adding a sink-vertex which is reachable from all these destinations. Then, the problem of checking the property "no black holes towards a (set of) box(es)/destination(s)", is lessened to checking that the sink-vertex is not reachable. The above invariants can also be used to check for (in)consistencies between the boxes' policies which are expected to be identical.

Expressivity. Anteater checks only safety properties that can be reduced to computing reachability of a remote network box (or more refined destination), based on reachability algorithms.

Experimentation. The evaluation was done with University of Illinois at Urbana-Champaign (UIUC) campus network: 178 routers (supporting predominantly OSPF, but also BGP and static routing), 70k end-clients and servers. The Anteater's performance is also stressed by checking the forwarding loop-freedom invariant in six autonomous system (AS) networks from [105].

Deployability. Anteater revealed 23 bugs in 2 hours, 7 runs in the UIUC network: 9 loops, 13 packet losses and 1

inconsistency. Scaling the number of routers on a campus network, the forwarding loop-freedom invariant checking time for a run ranges, in a roughly quadratic trend, from about 6 min, for a subset of 178 routers, to single-digit seconds (subset of two routers). It took about half an hour to check for the loop-free forwarding invariant property in a network of 384 nodes from the Rocketfuel project [105], which is the biggest one experimented in this paper.

Limitations. As the vertices in the graph represent network boxes (or destinations) and not global states of the network, the reachability analysis does not capture global behaviours but is limited to looking for reachable IP network addresses (routing reachability). Anteater is therefore limited to only three property categories: loop-free forwarding, connectivity and policy consistency of replicated boxes. Bugs which have no effect on the content of the FIBs cannot be caught by Anteater. Also, it might be the case that an inconsistent or incomplete view of the network is got by the Anteater, when the FIBs are updated while being retrieved. There is only one counterexample returned per verification attempt.

H. NetKAT [82] incomplete still

NetKAT is a domain-specific language and logic for specifying and verifying network packet-processing functions and part of the Frenetic [78] suite of languages.

Properties.

NetKAT provides a network programming language using predicates and policies. Any property that can be expressed as an equation or inequation can thus be checked. There is no result which properties can or can not be expressed. Examples show that reachability and isolation properties can be expressed.

Model. The main idea is to represent a network as an automaton to move packets. Kleene algebras describe the equational theory of regular expressions, whereas boolean algebras describe predicates, i.e. tests. KAT combines and unifies both and thus is able to describe network behaviour via the former concept and switch behaviour via the latter. The model is thus an extension of Kleene algebras with tests [108] (KAT).

The equational theory of NetKAT combines the axioms for KAT and those domain-specific axioms that describe the manipulation of packets. The equational theory is shown to be sound *and* complete wrt. the denotational (algebra) semantics. Syntactically, predicate expressions include constants true and false (as the latter predicate retains no packets, one uses the constant drop), matching a packet field f with value n ($f = n$) as well as negation, conjunction and disjunction. Packets are as usual modelled as records where field names are assigned values from a *finite* domain. Policy expressions include predicates, modifications for fields $f \leftarrow n$, sequential composition of policies p and q ($p; q$) as well as parallel composition ($p + q$), a policy of recording the current packet in the packet history (dup), and of course iteration (p^*). The histories are only used for reasoning purposes (and not required for computations).

For the algebraic equations there exists a denotational model that has been shown sound and complete. In this denotational model, every predicate and policy is interpreted as a function on packet histories, i.e. a function f that maps a given history h to a (possibly empty) set of histories $\{h_1, \ldots, h_n\}$. Returning the empty set models dropping a packet and its entire history. Returning a singleton set $\{h_1\}$ models modifying and forwarding a packet to a single location. Returning a set with more than one history models modifying a packet in several ways or forwarding it to several locations.

To model a network, one first models its topology, or better its behaviour, as parallel composition of link policies. Such a policy is the composition of a test whether a packet has arrived at the source link (switch and ingress port) and a modification that updates sw and port fields of the target link (i.e. switch and egress port). Let us use the letter t to denote topology policy.

Let p denote the forwarding policy of all the switches (parallel composition off individual switch policies). Then the network behaviour can be modelled as follows: packets are first processed by a switch then forwarded along the topology and then this process is repeated, so the end-to-end behaviour of the network is described by $(p; t)^*; p$. Normally, one identifies also the hosts where packets enter and leave the network. Writing in for the policy describing where packets enter the network and out for the policy describing where packers leave the network, one can describe the network by the expression $in; (p; t)^*; p; out$. If the history of packets is to be recorded, which is essential for reachability analysis, then one needs to model hop-by-hop processing using the following expression: $in; \mathsf{dup}; (p; t; \mathsf{dup})^*; p; out$.

Note that the expressions are in general richer than OpenFlow tables, so some compilation is necessary to implement the networks expressed in NetKAT.

Specification Language. The specification language is the language of algebra, so properties are expressed by equations or inequations between (network or policy) expressions. Let us assume network policy p and topology t and also assume a and b are policy expressions denoting (potentially specific kinds of) packets at end hosts a and b. Whether b is reachable from a can be formulated, e.g., as $a; \mathsf{dup}; (p; t; \mathsf{dup}^*; b \neq \mathsf{drop}$. The semantics of this expression can be proved to be exactly reachability. Similarly, one can express waypointing, access policies, loop freedom, or high-level operations such as network isolation (slicing)

Type of Check. Any property expressed as equation or inequation is checked by proving or disproving the algebraic equation in question.

Checking Phase. Checking is done statically by proving algebra equations. Whether the equation represents semantically the property of choice can be shown by using the denotational semantics.

Layer. NetKAT works on the network layer using policy expressions, there is no explicit representation of an SDN controller. High-level policies for switches can be expressed however. The flow table content of the switches is obtained

by compiling the (network) policy expression to a format that corresponds to flow table entries only.

Methodology. The network is modelled as an algebraic expression that one can reason about equationally. One can also compile such a NetKAT policy expression into a normal form, one that does not use the Kleene star (which is something that a switch cannot express semantically, as it is just a flow table) and that can be expressed by isolated switch policies that in turn can be normalised to expressions that basically correspond to flow table configuration policies (i.e. nested cascades of conditionals) which correspond to the OpenFlow standard. can be flow table in OpenFlow standard.

For the verification of properties algebraic equations need to be solved. It can be shown that the algebraic theory is decidable and PSPACE-complete. The actual check can be carried out in any tool that supports Kleene algebras with test. In [109] a coalgebraic semantics has been defined that allows for a more efficient verification of equations. It has been proved that the coalgebraic semantics coincides with the one described earlier.

Expressivity. NetKAT offers a policy language (like Frenetic [78]). The language design is guided by Kleene algebras with tests with basic primitives for networking. It thus comes with a denotational semantics and an equational theory that is sound and complete for this semantics. There is no equivalent of a controller program but the language principles allow unlimited formation of policies. This allows for reasoning about the network and particularly properties like reachability or loop-freedom. General temporal logic properties cannot be expressed in the original work but extensions are available that provide this [110].

Experimentation. The original NetKAT paper [82] does not present any practical experiments. The version with coalgebraic decision procedure [109] describes, however, a system that decides NetKAT equivalences. It is an OCaml program of about 4500 lines of code. This has been integrated into the Frenetic environment. The decision procedure uses the Brzozowski derivative of a set of strings S for a particular string u, defined as $u^{?1}S = \{v?\Sigma^* : uv \in S\}$. A famous results says that a string u is in a set of strings S denoted by a regular expressions if, and only if, the empty string $\epsilon \in u^{-1}S$. It is shown that the derivative can be encoded in matrix form for fast computation of the bisimulation equality by reducing the state space. Further optimisations using Hash consing and memoization and sparse multiplication (which avoids the numerous multiplications with 0 in a sparse matrix).

The following benchmarks have been tested: Topology Zoo [111], FatTrees that are generated manually, and Stanford Backbone [60]. Forwarding policies were introduced randomly between hosts in the first case, fat trees for a given pair of depth and fanout parameters were generated for the second case and the last is an explicit real world topology with 16 hosts. The Stanford backbone allows for comparison with HSA (see § V-B above). The properties checked were: connectivity between all pairs of hosts, loop-freedom, translation validation (i.e. does the compiler translate high-level policies into *equivalent* Open Flow forwarding rules?).

The results show that the given examples for Topology Zoo run within seconds for smaller examples and scale to hundreds of switches. The performance of the translation validation property is an order of magnitude slower (with thens of hours for topologies with thousands of switches) as it requires the full coalgebraic bisimulation algorithm to be run.

The FatTree benchmark tests also scalability. The performance is similar to the one observed for TopologyZoo where small networks verify in seconds, while larger ones can take many hours. On large inputs connectivity is fastest, loop freedom taking twice as long, and translation validation twice as long again.

For the last benchmark, the authors programmed a tool that translates router configurations of the Stanford backbone to NetKAT policies translating away prefix matching to use only concrete IP addresses in policies. Furthermore, to reduce the state space a static analysis was implemented that detects which packet fields in policies are static and then partial evaluates such policies to smaller ones before verification. With those improvements reachability queries run in the area of half a second which is comparable to the manually optimised version of HSA (not the original which is an order of magnitude slower).

Deployability. NetKAT decision procedure is integrated into the Frenetic toolkit. One can statically verify equivalence of network policies or translate a network policy into a set of open flow table descriptions. The performance for reachability appears to be comparable with optimised HSA and allows also absence of loops of reasonably large networks.

Limitations. Some limitations have been overcome by extensions: stateful NetKAT [112] to model switches with state, i.e. registers, Probabilistic NetKAT [113] to model uncertainty and randomized algorithms. Temporal NetKat [110] extends the language with linear temporal logic over finite traces.

I. NetSAT [114]

Properties. The properties considered in this work are Reachability between two ports, absence of forwarding Loops and Slice Isolation.

Model. The network topology is represented as a set of network elements (routers, NATs and firewalls) and links (pairs of ports). The header fields' values of a packet located at a specific port ($h_{switch,port}$) are abstracted to boolean values by a bit vector. The header encodes implicitly the switch and port id it is located at. A Boolean variable $valid(h_{switch,port})$ indicates the presence or absence of a valid packet at the port at issue. A switch is encoded as a function that relates packet header field values in the input and output ports. This function incorporates, as a prioritized list, all the rules extracted from the data structures of a network box, i.e., routing/forwarding tables - RIBs/FIBs in routers/switches, Access Control List (ACL) in firewalls, and Translation Table in NAT. Each rule, in turn, consist of a matching field and an action to matching one. A path that a packet takes through the network is represented as a conjunction of switch formulae, i.e., as assignments to the

header field variables. Thus, a single formula for the network is built. The network state is a mapping of network elements to the set of rules extracted from their data structures.

Specification Language. Properties are specified using propositional logic.

Type of Check. A SAT-based propositional logic verification approach which verifies static snapshots of data plane with a SAT solver.

Checking Phase. NetSAT is a static analyser: the network states are snapshots at a single instant in time which do not change during verification.

Layer. A framework for network forwarding data plane modeling and property checking.

Methodology. Zhang and Malik [114] use a Boolean Satisfiability Based Approach. They encode the data plane as a SAT formula and use combinational search to find errors. Using the single formula $\mathcal{N}$ for the network, which represents a single valid packet path, the satisfiability of $\mathcal{N} \wedge \neg \mathcal{P}$ indicates contradiction of property $\mathcal{P}$.

Expressivity. The framework is bound to express a set of three data plane properties.

Experimentation. Experiments are conducted using two sets of test benchmarks. The Stanford backbone network has 16 routers with VLAN tags, ACLs, etc., 15,000 rules which gives 6.2 million Conjunctive Normal Form variables and 32 million clauses. The other set of benchmarks are synthetically generated using the Waxman topology [115]. For the latter benchmarks, four subsets of experiments were run scaling the number of network boxes, rules and packet header size. Minisat [116] is used as the SAT solver for all the experiments.

Deployability. It took about 100 seconds to return satisfiable for both forwarding loop and reachability checking in Stanford network, and about 5 seconds for unsatisfiable (disproving) cases for reachability checking. A 50-switches topology, with 64-bit header length and $< 10^6$ rules, completed within 10 minutes, while checking a 190-switches topology with 160-bit headers (10^6 rules) took 4.5 hours. The size of the encoding is proportional to the number of routing rules/boxes which directly affects the execution time. The execution time, however, is less affected by the packet header length.

Limitations. As a static technique, NetSAT considers a single snapshot of the network and assumes that the network is stateless. Only one satisfying assignment is computed each time.

J. Kinetic0 (early version[11]) [117]

Properties. The properties of interest in this paper are those expressible as set of packets and their trajectories, referred to as *trace properties*. Such can be access control, connectivity, routing correctness, loop-freedom, 'no black holes', blacklisting, correct VLAN tagging, and waypointing. The substantive objective of [117], though is an update abstraction which is guaranteed to preserve these properties when transitioning between configurations; or in other words, that every packet traversing the network is consistently processed by one and only forwarding policy. It is provided a library of prevalent network properties, such as loop-freeness, but custom properties can be added as well.

Model. A so-called located packet lp is modelled as a bit vector along with a switch port where it is located. For each packet, a trajectory t is associated with, which consists of an array of ports the packet has passed through. A switch function S maps a located packet to a (set of) located one(s). The topology function T is encoded as a permutation of the set of ports. The switch and topology functions constitute the network configuration C. A network state N is a pair consisting of: (1) a snapshot, at a particular instance of the transition relation, of the mapping Q from all the switch ports to the set of packets (updated with their hops history) enqueued in the port queue, and (2) the network configuration C. The network updates are alterations to the switch function. The update transition is encoded as an overwriting of the switch function with the new mappings between located packets. The semantics of the network is a transition system whose transition relation is an ordered concatenation of all atomic update-transitions (called update sequence us). The notation $N \xrightarrow{us}{}^{*} N'$ is used to denote the switch of the transition system from N to N' due to the execution of a list us of control messages from the SDN controller to the forwarding plane. [12] In order to develop a run-time mechanism, the per-packet consistency abstraction is presented which specifically focuses on the network configuration changes. In the per-packet consistency abstraction, when an update is applied to network, all packets are processed by either the pre-update configuration or the post-update one. This "2-phase update" algorithm is based on a configuration versioning where packets are tagged with a configuration version id (using the VLAN field). The idea is as follows: first, the switches in the middle of the network (i.e., switches that do not provide an entry point into network) are updated with the new configuration rules, tagged with the new version so that they can be hit only by packets tagged with same version. The old configuration is left in place. Next, the new configuration is installed in the edge switches. The edge rules stamp all ingress packet with the new version number. And last, the old configuration rules are removed from all the switches once all the packets hitting them are drained out of the network. The per-flow consistency is another abstraction which is a generalisation of the per-packet consistency one. In this abstraction, all packets of a flow are processed by the same configuration version.

Specification Language. Branching time temporal logic (CTL) is used here to specify behaviours along paths a packet is allowed to walk on.

Type of Check. The updates proposed in [117] are provably consistent: guarantees are provided (proofs) that the abstractions (per-packet/flow consistent updates) preserve all trace

[11] Kinetic, which is part of the frenetic [78] family, is also quoted and exclusively presented in a later paper [81] but positioned there in a more specific role: that of a domain specific language (DSL).

[12] $\rightarrow^{*}$ is the reflexive and transitive closure of the transition relation $\rightarrow$.

properties, i.e., if any property holds prior to, as well as after an update, then the property also holds in-between.

Checking Phase. Since the semantics include updates, a dynamically evolving network is modelled.

Layer. In order to pre-process the rules (so that they tag all packets entering the network with a version number), manage and refine them dynamically over time, [117] is placed on top of the SDN controller in the capacity of a run-time system.

Methodology. The abstractions proposed here can be (theoretically) explored by any static analysis approach to verify the invariant trace properties as network configurations evolve. However, model checking is used by the authors to demonstrate the idea.

Expressivity. Properties can be expressed in terms of the path(s) traversed by a packet or sets of packets belonging to the same flow. Branching time temporal logic (CTL) is used to specify the allowed paths, which has been shown to be adequate for expressing such properties.

Experimentation. [117] was evaluated through a set of simple experiments which were developed using Mininet [106]. For the main abstraction, i.e., per-packet consistency, two network applications are implemented: routing and multicast. The former computes the shortest paths between all hosts and updates routes as hosts get online/offline and switches up/down. The latter, groups the hosts into two multicast clusters[13]. The hosts in each cluster are connected into a spanning tree. Both applications are run in three different scenarios: a) adding/removing randomly 10% – 20% of the hosts, b) re-routing randomly 20% of the routes (simulating switch removals), and c) both at the same time. Three different topologies are used for each scenario: fat-tree [118], small-world [119] and Waxman [115]. Each topology contains 192 hosts and 48 switches. For the multicast example, one of the multicast groups is changed each time. The evaluation of the per-flow updates, is done through a load-balancer that divides traffic between two server replicas. The update for this experiment involves bringing new server replicas online and re-balancing the load.

Deployability. The update abstractions are implemented in a system called Kinetic, in Python, which sits on top of NOX controller [107]. Both abstractions are, from implementation point of view, represented by a function which implements the update transition for any new configuration. To unroll the transition system Kinetic uses the NuSMV model checker [120] as verification engine. For reducing the overhead required to perform consistent updates and achieve better performance, several optimisations are introduced. The idea behind the optimisations is that instead of deploying a full 2-phase update mechanism that installs the full new policy and then uninstalls the old policy, only the "delta" between the two configurations is considered. These optimisations (referred to as *pure extensions/retractions*) are applicable under certain conditions – e.g., when the update only affects a subset of switches, rules or network traffic as hosts come online or go offline – and as such, since a complete two-phase update is not required, the transition to the new configurations is achieved in less time. The update cost is proportional to the number of rules that changed between configurations, as opposed to unoptimised (full 2-phase) update, where the cost is proportional to the size of the entire new configuration. For the multicast application, the subset optimisations yield fewer improvements, as almost all routes change when the spanning tree changes, bringing on an expensive update. The optimisations are not applied for the per-flow mechanism, therefore no optimisation evaluation of the load balancer.

Limitations. The main drawback of [117] is that it requires that both versions of rule-sets are at the same time represented on the switches, resulting, in the worst case (when the subset optimization is not applicable and a full two-phase update takes place), in double the TCAM storage capacity overhead. Another factor which increases the rule-space overhead is the tag-matching. The properties that one can check in [117] are limited to those expressible as sequence of hops a packet has traversed.

[13] A multicast cluster is a set of hosts that listen for and receive traffic addressed to a special, shared multicast IP

K. Katta [121]

Properties. Same as [117], the property set consists of traversal path invariants which are checked whether they are obeyed throughout the execution of an update in the network.

Model. A location is a pair of switch id and an ingress port. The rules are modelled as 3-tuples, each consisting of: i) a predicate P on ingress packets, encoded by a pattern for matching header fields and a location the matched packet arrived on, ii) an action a on matching, and iii) a priority z used to pick unambiguously a rule among rules with overlapping patterns. Hence, a predicate is a symbolic ingress flow. A global network policy R is a set of rules. The policies are versioned using the VLAN or MPLS header fields. The new forwarding policy which is about to be installed, is sliced into sub-policies (subset of rules) by means of predicates. Each sub-policy defines a sub-flow. This slicing allows the new policy to be applied in rounds. Each round is executed pursuant to the 2-phase paradigm [117]. In a 2-phase update, a policy acts differently on edge switches than it does on middle ones: at ingress edge switches, the rules stamp all packets entering the network with a version number; the newly written version-field serves as a new matching field for the rules at the middle switches; and finally the rules at egress switches remove the version information from the packets.

Specification Language. Same as [117].

Type of Check. Same as [117].

Checking Phase. Same as [117].

Layer. Same as [117].

Methodology. The abstraction in [121], wraps the one in [110] and, moreover, adjusts it aiming at tackling rising TCAM space, caused by the coexistence of both new and old policies at the same switch, when implementing consistent updates. To achieve this, the 2-phase update protocol [117] is applied in instalments. This, of course, sacrifices updating time to cut down space. It, first, divides the global policy into a set of

consistent slices, and next applies the 2-phase update [117] for all the individual configuration slices one by one. Given a predicate P_i, the first algorithm in [121] computes the slice i as the set of all rules that match the flows asserted by P_i. To determine in each round the new rules that should be placed in the network, but and the the old rules that can safely be removed in order to preserve consistency, the algorithm analyses and keeps track of the dependencies between flows in the old and new policies. A rule from the old policy remains active at the switches as long as there is some in-flight flow which can be matched by it. More formally, if P_t is a predicate which defines a subset of new rules installed up to a transitioning instance t, then any old rule r_o falling under the predicate $\neg P_t$ should not be removed as long as there is an in-flight flow in the network which can be matched by r_o. The computation of the flows which correspond to a sub policy is done through a reachability analysis along the lines of [60].
While the first algorithm only generates the slice given a predicate, another algorithm is presented in [121] in order to decide: i) how many slices the policy should be split into, and ii) which predicates are to be used at each round, given a predefined number K of slices (rounds) for an update. The latter problem can be reformulated as an optimisation problem as follows: "Partition the set of all ingress predicates into K ordered subsets, $optimally$", where $optimally$ is quantified by capping the worst-case rule-space overhead. Reasonably, this is posed as a mixed-integer linear program (MILP). The MILP finds the minimum rule-space overhead subject to several constraints imposed by the semantics of the consistent updates. The optimisation algorithm evidently aims to minimise the rule-space overhead.
Expressivity. Same as [117].
Experimentation. The experiments are performed on the three popular classes of topologies used in [117] (Fattree, Small-world and Waxman), each with 24 switches and 576 hosts. The scenario the experiments run, uses two load-balancing policies, swapping them over. The load-balancing policy allots randomly a set of server replicas. Then, it intercepts the ingress flows in the edge switches and, by modifying their destination IP address, distributes them randomly among the server replicas. The second load-balancing policy can be obtained from the first by cutting back the number of replicas.
Deployability. Preliminary empirical results are presented in [121]. The Gurobi [122] optimiser is used as optimization solver for the mixed-integer linear programming. It returns the optimal solution in a range of few seconds for most runs; finding the exact optimal solution, however, lasts much longer (hours). In addition to the number of slice capping, the MILP can also cap the rule-space overhead threshold per switch and minimise the total time required to complete an update.
In the experiment using load-balancing, about 100 OpenFlow rules are deployed at each switch. Each rule has exact-matching on source and destination IP addresses (i.e., covering the entire input string). A 6-round incremental update can reduce the space overhead by a factor of ten. When the rule-space overhead is capped at 5%, then about 20% of flows are left to be processed by the old policy by the end of the 1st round, and only about 1% by the end of the 3rd round.
Limitations. Bounding the worst-case rule-space overhead comes at a price of slower update.

L. Minesweeper [90]

Properties. Minesweeper encodes the behaviour of the network and a (negated) property of interest into a system of SMT constraints. Properties can therefore be about the encoded behaviour of the network. Additional variables can be added to routers and their interfaces to reason about more complex aspects of network operation. Example properties include reachability, isolation, loop freedom, black holes, waypointing (i.e. traffic traversing a specific chain of devices), equal path length, disjoint paths. Minesweeper encodes behaviour of routing protocols (e.g. BGP, OSPF) into SMT constraints, therefore properties can be about routing protocol aspects; e.g. neighbour or path preference, equivalence of configuration of routers and load balancing.
Model. Minesweeper's modelling philosophy is based on the fact that the network plane solves the stable paths problem [123], and models the network behaviour into SMT constraints so that satisfiable assignments correspond to stable paths in the control plane. With this approach, Minesweeper captures all possible stable paths. Constraints that describe properties of interest are then added to perform verification. Minesweeper can reason about data packets in properties using integer variables for the source and destination IP address and port and the protocol header. Packet re-writing is not allowed therefore there is a single global variable in the used formula.

Minesweeper models a router's behaviour as a set of routing protocols that operate independently to each other, exchanging routing information with other protocols internally and with other routers. The objective of each router is to select a best route for a specific destination prefix based on the information it receives from local and remote protocols instances. Protocol instances exchange protocol messages which are modelled as records of symbolic values. Minesweeper defines a number of values (concrete or unconstrained) that can be used to model messages exchanged by routing protocols (e.g. routing prefix, prefix length, distance, BGP local preference etc.).

Routing information flow is modelled as a graph (which is built on top of a given network topology) interconnecting the various protocols instances within and across routers. Edge labels *e* and *i* indicate a message exported by a routing protocol instance and the same message after being processed by an incoming filter at the destination instance, respectively. Minesweeper verifies a property with respect to a specific symbolic packet (modelled as discussed above), therefore it only considers routing prefixes (for all protocol instances) that are relevant to the symbolic packet. Import filters operate on incoming protocol messages and can either drop them or alter any of the protocol fields.

Each protocol instance selects the best route for given IP prefixes by ordering all available routes in a protocol-specific fashion (e.g. BGP prefers the route with the highest

administrative distance). Each router will install a single route in its data plane which is selected to be the best route offered by each locally running protocol instance. After selecting a best route, each protocol instance exports messages to all its peer protocol instances; these can be pre-processed by an export filter. Finally, Minesweeper encodes access control lists in the data plane as constraints on routing entries used to decide on the outgoing interface used to route a symbolic packet.

Additional variables can be used to instrument Minesweeper's model so that a desired property can be verified. For example, reachability can be verified by instrumenting the model with a *reach* variable at each router indicating whether the router can reach a specific subnet.

Specification Language. Minesweeper models the control and data plane as SMT constraints written in first-order logic.

Type of Check. Minesweeper can verify properties for all data planes (i.e. using symbolic packets) and any routing protocols that can be modelled as discussed above. Minesweeper can calculate all stable sets given a specific (distributed) routing configuration and searches for just one of the stable sets (i.e. control plane computation) that violates a given property. It will not find all violations (in different stable sets - control plane computations) at once; instead one can pinpoint a bug, fix it and subsequently search for the next one. In the absence of bugs, Minesweeper can verify the correctness of a configuration for all control and data planes

Checking Phase. Minesweeper is a static analysis tool that uses Batfish [88] to parse vendor-specific router configurations which then translates into the symbolic model discussed above. Property verification is done by the Z3 SMTP solver.

Layer. Minesweeper models both the data plane (using symbolic packets and data plane filters) and the control plane (using protocols instances that exchange routing messages among each other and incoming filters on these messages). As a result, it can check the correctness of routing configurations for all modelled control planes and all data planes.

Methodology. Given the employed network model, Minesweeper relies on an SMT solver to verify the correctness of given properties or identify a bug for the given system of SMT constraints (including the property to be checked).

Expressivity. Minesweeper allows for model instrumentation using user-defined variables that can be integrated into the SMT constraints that are passed to the solver. In that sense, Minesweeper is as expressive as first-order logic allows it to be. Control plane modelling is extensive and the authors provide a list of protocols/routing functionality that can be modelled by Minesweeper out of the box.

Experimentation. The authors evaluated Minesweeper in relatively small real-world topologies with real configurations and synthetic, but functional, topologies of various sizes. Example properties they checked include reachability to management interfaces, local equivalence of routers and absence of black holes. Minesweeper revealed tens of property violations for all the aforementioned properties. Minesweeper verified all properties for the small real-world networks (2 to 25 routers each) in under a second (in the majority of cases). As expected, verification is slower as the size of the network (and the respective lines of routing configuration) increases. Scalability analysis showed that for synthetic network configurations, Minesweeper's verification performance is in the order of minutes or tens of minutes. Finally, the authors assessed the performance gains from the proposed optimisations; replacing bit vectors for advertised prefixes speeds up verification by over 200x on average. The proposed slicing optimizations result in performance gains of 2.3x on average.

Deployability. Minesweeper is deployable as long as the deployed routing protocols are part of the modelled control plane functionality (e.g. BGP, OSPF, static routes etc.). It parses vendor-specific routing configurations (using Batfish [88]), which are then transformed to the defined model. The authors applied Minesweeper on configurations of real-world networks that have been operational for years. The authors propose a number of optimisations that make Minesweeper deployment realistic. Prefix elimination is based on the fact that prefixes in routing messages do not need to be represented explicitly because the destination IP address of the symbolic packet and the prefix length are known, therefore there is a unique valid corresponding prefix for that destination IP. With this optimisation there is no need to represent prefixes with bit vectors which are very expensive for SMT solvers. Loop detection for routing messages of policy-based routing protocols is also expensive if state is to be maintained in routers, therefore the authors propose to use protocol-specific information (e.g. for BGP) to ensure that loops will never occur. Finally, the authors propose a number of 'network slicing' optimisations, that remove bits from the encoding that are unnecessary for the final solution; e.g. if BGP routers never set a local preference, then the local preference attribute will never affect the decision and is removed.

Limitations. Minesweeper can only reason about the stable sets to which the control plane will converge, therefore it cannot verify properties during the transition of routing protocols' state; a simulation tool would be needed for that purpose but the authors note that this compromise significantly improves performance. Second, Minesweeper only considers elements of the control plane that influence the forwarding decisions pertaining to a single symbolic packet at a time (through the defined *valid* field that checks if a message is advertised from a neighbor and not filtered and the control plane destination prefix applies to the destination IP of this symbolic packet). It is therefore expensive to model features that introduce dependencies among destinations. Minesweeper is a static analysis tool therefore if the routing configuration changes, properties must be verified from scratch. Finally, Minesweeper, cannot reveal all bugs at once; a bug first needs to be fixed in order to continue with the verification process.

VI. Conclusion and Future Directions

In this review, the evidence for the two sides of the SDN verification coin has been exhibited; that it is reasonably

practicable to enhance network verification taking advantage of the goodies included in the SDN package, and that the SDN architecture introduces new vulnerabilities that are not present in traditional networks.

In summary, the fate of future networks is still unknown. Programmatic networks are still maturing and largely untested. Early implementations of every protocol have been buggy – SDN is no different. Although the work on advancements to network verification is more mature than ever, so much work has been purely theoretical, or at best, tested on unrealistically small and simple problems – let alone battle tested or applied on a model of the system and not the actual system. Many research in this area attempt to verify each SDN component separately or focus on domain specific heuristics, but do not provide a unified approach to automatically validate the logical consistency between all the compartments of the ecosystem. A cross-layer modelling and verification system that can analyse the configurations and policies across both application and network components as a single unit is still to come.

REFERENCES

[1] N. McKeown, "Mind the Gap," in *SIGCOMM Keynote*, 2014.

[2] K. Greene, "TR10: Software-defined networking," *MIT Technology Review*, 2009. [Online]. Available: http://www2.technologyreview.com/article/412194/tr10-software-defined-networking/

[3] ONF, "SDN Architecture Overview," *Technical Report*, 2013. [Online]. Available: https://tinyurl.com/kl5gd5m

[4] ——, "Software-defined networking: The new norm for networks," *White Paper*, 2012.

[5] "ITU-T Y.3300:Framework of software-defined networking."

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, 2008.

[7] Open Networking Foundation, "SDN Architecture," *Onf*, 2014. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf

[8] P. B. N. Bosshart, D. I. Daly *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, 2014.

[9] M. Al-Fares, S. Radhakrishnan, and B. Raghavan, "Hedera: Dynamic Flow Scheduling for Data Center Networks." in *NSDI*, 2010.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: scaling flow management for high-performance networks," *SIGCOMM*, 2011.

[11] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proceedings - IEEE INFOCOM*, 2011.

[12] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: fine grained traffic engineering for data centers," in *ACM CoNEXT*, 2011.

[13] R. Trestian, "MiceTrap: Scalable traffic engineering of datacenter mice flows using OpenFlow," *2013 IFIP/IEEE International Symposium on Integrated Network Management*, 2013.

[14] S. Jouet, C. Perkins, and D. Pezaros, "OTCP: SDN-managed congestion control for data center networks," in *Proceedings of the NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, 2016.

[15] R. Van Der Pol, S. Boele, F. Dijkstra, A. Barczyk, G. Van Malenstein, J. H. Chen, and J. Mambretti, "Multipathing with MPTCP and open flow," in *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, 2012.

[16] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing web traffic using OpenFlow," *SIGCOMM*, 2009.

[17] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-Based Server Load Balancing Gone Wild Into the Wild : Core Ideas," *Hot-ICE'11 Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and servicesworks and services*, p. 12, 2011.

[18] A. F. Trajano and M. P. Fernandez, "Two-phase load balancing of In-Memory Key-Value Storages through NFV and SDN," in *Proceedings - IEEE Symposium on Computers and Communications*, 2016.

[19] F. Carpio, A. Engelmann, and A. Jukan, "DiffFlow: Differentiating short and long flows for load balancing in data center networks," in *2016 IEEE Global Communications Conference, GLOBECOM 2016 - Proceedings*, 2016.

[20] M. Alizadeh, N. Yadav *et al.*, "CONGA," in *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14*, 2014.

[21] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable Load Balancing Using Programmable Data Planes," in *ACM Symposium on SDN Research (SOSR)*, 2016.

[22] Y. Li and D. Pan, "OpenFlow based Load Balancing for Fat-Tree Networks with Multipath Support," in *Proc. 12th IEEE International Conference on Communications (ICC'13)*, 2013.

[23] H. Hu, G.-J. Ahn, W. Han, and Z. Zhao, "Towards a Reliable SDN Firewall," in *ONS*, 2014.

[24] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FlowGuard: Building Robust Firewalls for Software-Defined Networks," *HotSDN*, 2014.

[25] N. P. Katta, J. Rexford, and D. Walker, "Logic Programming for Software-Defined Networks," *Workshop on Cross-Model Design and Validation (XLDI), ACM*, 2012.

[26] J. Wang, Y. Wang, H. Hu, Q. Sun, H. Shi, and L. Zeng, "Towards a security-enhanced firewall application for openflow networks," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013.

[27] G. Yao, J. Bi, and P. Xiao, "Source address validation solution with OpenFlow/NOX architecture," in *Proceedings - International Conference on Network Protocols, ICNP*, 2011.

[28] N. Feamster, J. Rexford, S. Shenker, R. Clark, R. Hutchins, D. Levin, and J. Bailey, "SDX: A software-defined Internet exchange," *Open Networking Summit*, 2013.

[29] L. Liu, T. Tsuritani, I. Morita, H. Guo, and J. Wu, "Experimental validation and performance evaluation of OpenFlow-based wavelength path control in transparent optical networks," *Optics Express '11*.

[30] L. Liu, D. Zhang *et al.*, "Field trial of an openflow-based unified control plane for multilayer multigranularity optical switching networks," *Journal of Lightwave Technology*, 2013.

[31] S. Azodolmolky, R. Nejabati, E. Escalona, R. Jayakumar, N. Efstathiou, and D. Simeonidou, "Integrated OpenFlow–GMPLS control plane: an overlay model for software defined packet over optical networks," *Optics Express*, 2011.

[32] M. Channegowda, R. Nejabati *et al.*, "Experimental demonstration of an OpenFlow based software-defined optical network employing packet, fixed and flexible DWDM grid technologies on an international multi-domain testbed," *Optics Express*, 2013.

[33] A. Sadasivarao, S. Syed, P. Pan, C. Liou, A. Lake, C. Guok, and I. Monga, "Open Transport Switch: A Software Defined Networking Architecture for Transport Networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, 2013.

[34] Y. Yiakoumis, K.-K. Yap, S. Katti, G. Parulkar, and N. McKeown, "Slicing home networks," *HomeNets '11*.

[35] R. Mortier, T. Rodden, T. Lodge, D. McAuley, C. Rotsos, A. W. Moore, A. Koliousis, and J. Sventek, "Control and understanding: Owning your home network," in *2012 4th International Conference on Communication Systems and Networks, COMSNETS 2012*, 2012.

[36] R. B. Abdallah, T. Risset *et al.*, "SoftRAN: Software defined radio access network," *IEEE Communications Magazine*, 2014.

[37] T. Chen, H. Zhang, X. Chen, and O. Tirkkonen, "SoftMobile: Control evolution for future heterogeneous mobile networks," *IEEE Wireless Communications*, 2014.

[38] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, "SoftCell: Scalable and Flexible Cellular Core Network Architecture," in *the ninth ACM conference on Emerging networking experiments and technologies*, 2013.

[39] L. E. Li, Z. M. Mao, and J. Rexford, "Toward software-defined cellular networks," in *Proceedings - European Workshop on Software Defined Networks, EWSDN 2012*, 2012.

[40] J. Liu, S. Zhang, N. Kato, H. Ujikawa, and K. Suzuki, "Device-to-device communications for enhancing quality of experience in software defined multi-tier LTE-A networks," *IEEE Network*, 2015.

[41] I. F. Akyildiz, P. Wang, and S. C. Lin, "SoftAir: A software defined networking architecture for 5G wireless systems," *Computer Networks*, 2015.

[42] H. Zhang, S. Vrzic, G. Senarath, N. D. Dào, H. Farmanbar, J. Rao, C. Peng, and H. Zhuang, "5G wireless network: MyNET and SONAC," *IEEE Network*, 2015.

[43] V. Yazici, U. C. Kozat, and M. O. Sunay, "A new control plane for 5G network architecture with a case study on unified handoff, mobility, and routing management," *IEEE Communications Magazine*, 2014.

[44] M. Bansal, J. Mehlman, S. Katti, and P. Levis, "OpenRadio: A Programmable Wireless Dataplane," in *Proceeding HotSDN '12 Proceedings of the first workshop on Hot topics in software defined networks*, 2012.

[45] M. Yang, Y. Li, D. Jin, L. Su, S. Ma, and L. Zeng, "OpenRAN: A Software-defined Ran Architecture via Virtualization," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM - SIGCOMM '13*, 2013.

[46] C. J. Bernardos, A. De La Oliva, P. Serrano, A. Banchs, L. M. Contreras, H. Jin, and J. C. Zúñiga, "An architecture for software defined wireless networking," *IEEE Wireless Communications*, 2014.

[47] H. Ali-Ahmad, C. Cicconetti, A. De La Oliva, V. Mancuso, M. R. Sama, P. Seite, and S. Shanmugalingam, "An SDN-based network architecture for extremely dense wireless networks," in *SDN4FNS 2013 - 2013 Workshop on Software Defined Networks for Future Networks and Services*, 2013.

[48] K.-K. Yap, R. Sherwood, M. Kobayashi, T.-Y. Huang, M. Chan, N. Handigol, N. McKeown, and G. Parulkar, "Blueprint for introducing innovation into wireless mobile networks," in *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures - VISA '10*, 2010.

[49] T. Luo, H. P. Tan, and T. Q. S. Quek, "Sensor openflow: Enabling software-defined wireless sensor networks," *IEEE Communications Letters*, 2012.

[50] P. Dely, A. Kassler, and N. Bayer, "OpenFlow for wireless mesh networks," in *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, 2011.

[51] I. Ku, Y. Lu, M. Gerla, R. L. Gomes, F. Ongaro, and E. Cerqueira, "Towards software-defined VANET: Architecture and services," in *2014 13th Annual Mediterranean Ad Hoc Networking Workshop, MED-HOC-NET 2014*, 2014.

[52] M. A. Salahuddin, A. Al-Fuqaha, and M. Guizani, "Software-defined networking for rsu clouds in support of the internet of vehicles," *IEEE Internet of Things Journal*, 2015.

[53] S. Jain, M. Zhu *et al.*, "B4: Experience with a Globally-Deployed Software Defined WAN," in *SIGCOMM*, 2013.

[54] P. Patel, D. Bansal *et al.*, "Ananta: Cloud Scale Load Balancing," *SIGCOMM*, 2013.

[55] "Nicira- It's time to virtualize the network." [Online]. Available: http://www.netfos.com.tw/PDF/Nicira/ItisTimeToVirtualizetheNetworkWhitePaper.pdf

[56] S. Natarajan, A. Ramaiah, and M. Mathen, "A Software defined Cloud-Gateway automation system using OpenFlow," in *Proceedings of the 2013 IEEE 2nd International Conference on Cloud Networking, CloudNet 2013*, 2013, pp. 219–226.

[57] "Open Networking Foundation." [Online]. Available: https://www.opennetworking.org/

[58] "Open Daylight." [Online]. Available: http://www.opendaylight.org/

[59] Open Networking Foundation, "OpenFlow Switch Specification 1.5.1," Tech. Rep., 2015. [Online]. Available: https://www.opennetworking.org/images//openflow-switch-v1.5.1.pdf

[60] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, 2012.

[61] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-wide Invariants in Real Time," in *NSDI*, 2013.

[62] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *SIGCOMM*, 2011.

[63] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in *NSDI*, 2013.

[64] R. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury, "A verification platform for SDN-enabled applications," in *IC2E '14*.

[65] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*, 2012, p. 55.

[66] C. Killian, C. Killian, J. W. Anderson, J. W. Anderson, R. Jhala, R. Jhala, A. Vahdat, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code," in *NSDI*, 2007.

[67] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *SafeConfig*, 2010.

[68] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," *IEEE/ACM Transactions on Networking*, vol. 22, no. 2, pp. 554–566, 2014.

[69] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat, "Building distributed systems using Mace," in *IEEE P2P'09 - 9th International Conference on Peer-to-Peer Computing*, 2009, pp. 91–92.

[70] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace," *ACM SIGPLAN Notices*, vol. 42, no. 6, p. 179, 2007.

[71] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test Openflow Applications," in *NSDI*, 2012.

[72] R. Majumdar, S. Deep Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in *FMCAD*, 2014.

[73] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "VeriCon," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*, 2013, pp. 282–293.

[74] ——, "VeriCon: Towards Verifying Controller Programs in Software-defined Networks," in *PLDI*, 2014.

[75] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev, "SDNRacer: concurrency analysis for software-defined networks," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 402–415, 2016.

[76] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev, "SDNRacer: Detecting Concurrency Violations in Software-defined Networks," *Sosr*, pp. 22:1–22:7, 2015.

[77] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A balance of power: Expressive, Analyzable Controller Programming Tim," *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, 2013.

[78] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming - ICFP '11*, vol. 46, no. 9, 2011, p. 279.

[79] "The Frenetic Research Project." [Online]. Available: http://www.frenetic-lang.org

[80] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pp. 1–14, 2013.

[81] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable Dynamic Network Control," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 59–72.

[82] C. J. Anderson, N. Foster, A. Guha, J. B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetkAT: Semantic foundations for networks," in *POPL*, 2014.

[83] A. Guha, M. Reitblatt, and N. Foster, "Machine-verified network controllers," *ACM SIGPLAN Notices*, vol. 48, no. 6, p. 483, 2013.

[84] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '12*, 2012, p. 217.

[85] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *Proceedings of the first workshop on Hot topics in software defined networks*, 2012, pp. 43–48.

[86] A. Horn, A. Kheradmand, and M. R. Prasad, "Delta-net: Real-time Network Verification Using Atoms," in *NSDI*, 2017.

[87] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, 2016.

[88] A. Fogel, S. Fung *et al.*, "A General Approach to Network Configuration Analysis," *NSDI*, 2015.

[89] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *POPL*, 2016.

[90] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A General Approach to Network Configuration Verification," in *sigcomm*, 2017.

[91] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Systems*, vol. 2, no. 4, pp. 255–299, 1990.

[92] R. Alur and T. A. Henzinger, "A really temporal logic," *Journal of the ACM*, vol. 41, no. 1, pp. 181–203, 1994.

[93] R. Alur, T. Feder, and T. a. Henzinger, "The benefits of relaxing punctuality," *Journal of the ACM*, vol. 43, no. 1, pp. 116–146, 1996.

[94] A. Nayak and A. Reimers, "Resonance: dynamic access control for enterprise networks," *Wren*, pp. 11–18, 2009.

[95] Z. A. Qazi, C. C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *SIGCOMM*, 2013.

[96] B. Bingham, J. Bingham, F. M. De Paula, J. Erickson, G. Singh, and M. Reitblatt, "Industrial strength distributed explicit state model checking," in *PDMC*, 2010.

[97] D. L. Dill, "The Murϕ verification system," in *CAV*, 1996.

[98] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[99] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella, "Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud," 2014. [Online]. Available: arxiv:1305.0209[cs.NI]

[100] L. De Moura and N. Bjørner, "Z3: An efficient SMT Solver," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008.

[101] J. C. Mitchell, T. L. Hinrichs, N. S. Gude, M. Casado, and S. Shenker, "Practical declarative network management," 2009.

[102] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, 1997.

[103] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[104] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Communications of the ACM, Magazine*, 2013.

[105] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring ISP Topologies With Rocketfuel," *IEEE/ACM Transactions on Networking*, 2004.

[106] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*, 2010, pp. 1–6.

[107] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *SIGCOMM Computer Communication Review*, vol. 38, no. 3, p. 105, 2008.

[108] D. Kozen, "Kleene Algebra with Tests," *TOPLAS*, 1997.

[109] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson, "A coalgebraic decision procedure for NetKAT," in *POPL*, 2015.

[110] R. Beckett, M. Greenberg, and D. Walker, "Temporal NetKAT," in *PLDI*, 2016.

[111] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE*, 2011.

[112] J. McClurg, H. Hojjat, N. Foster, and P. Cěrnyá, "Event-driven network programming," in *PLDI*, 2016.

[113] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic NetKAT," in *ESOP*, 2016.

[114] S. Zhang and S. Malik, "SAT based verification of network data planes," in *Automated Technology for Verification and Analysis. Springer*, 2013.

[115] B. M. Waxman, "Routing of Multipoint Connections," *IEEE Journal on Selected Areas in Communications*, 1988.

[116] N. Eén and N. Sörensson, "An Extensible SAT-solver," 2010.

[117] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM*, 2012.

[118] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," in *IEEE Transactions on Computers*, 1985.

[119] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," in *The Structure and Dynamics of Networks*, 2011.

[120] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," 2002.

[121] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *HotSDN 2013 - Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[122] GUROBI Optimization Inc, "Gurobi Optimizer reference manual," Tech. Rep., 2018. [Online]. Available: http://www.gurobi.com

[123] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *IEEE/ACM Transactions on Networking*, 2002.

Chapter 3

Towards Model Checking Real-World Software-Defined Networks

This chapter is an extended version of the author's paper "Towards Model Checking Real-World Software-Defined Networks" in Proceedings of the 32nd International Conference on Computer-Aided Verification (CAV), 2020, and has been reproduced here with the permission of the copyright holder. The chapter constitutes the core piece of the thesis introducing a coherent yet optimised and highly expressive computational SDN model (code-named MoCS). MoCS is based on an interleaving semantics where concurrency of actions is reduced to the non-deterministic choice among their possible sequentialisations, allowing for capturing complex (dependency) patterns among events. To keep the computational cost manageable, MoCS explores systematically possibilities for optimisation by identifying independent and invisible for the property actions (also called safe actions) in a context-aware manner. MoCS's performance is measured using three examples of network controllers: a stateless and a stateful firewall, and a MAC learning application. As we scale up the network, we investigate the behaviour of MoCS in terms of verification throughput, number of visited states and required memory. We show that (1) describing an SDN in a more complicated semantics, while (2) devising the right optimisations, subtle real-world bugs can be discovered using model checking without sacrificing performance.
This chapter also sets the baseline for later phases of modelling (Chapter 4).

Towards Model Checking Real-World Software-Defined Networks

(version with appendix)

Vasileios Klimis, George Parisis, and Bernhard Reus

University of Sussex, UK
{v.klimis, g.parisis, bernhard}@sussex.ac.uk

Abstract. In software-defined networks (SDN), a controller program is in charge of deploying diverse network functionality across a large number of switches, but this comes at a great risk: deploying buggy controller code could result in network and service disruption and security loopholes. The automatic detection of bugs or, even better, verification of their absence is thus most desirable, yet the size of the network and the complexity of the controller makes this a challenging undertaking. In this paper, we propose MOCS, a highly expressive, optimised SDN model that allows capturing subtle real-world bugs, in a reasonable amount of time. This is achieved by (1) analysing the model for possible partial order reductions, (2) statically pre-computing packet equivalence classes and (3) indexing packets and rules that exist in the model. We demonstrate its superiority compared to the state of the art in terms of expressivity, by providing examples of realistic bugs that a prototype implementation of MOCS in Uppaal caught, and performance/scalability, by running examples on various sizes of network topologies, highlighting the importance of our abstractions and optimisations.

Note: This is an extended version of our paper (with the same name), which appears in CAV 2020.

1 Introduction

Software-Defined Networking (SDN) [16] has brought about a paradigm shift in designing and operating computer networks. A logically centralised controller implements the control logic and 'programs' the data plane, which is defined by flow tables installed in network switches. SDN enables the rapid development of advanced and diverse network functionality; e.g. in designing next-generation inter-data centre traffic engineering [10], load balancing [19], firewalls [24], and Internet exchange points (IXPs) [15]. SDN has gained noticeable ground in the industry, with major vendors integrating OpenFlow [36], the de-facto SDN standard maintained by the Open Networking Forum, in their products. Operators deploy it at scale [27,37]. SDN presents a unique opportunity for innovation and rapid development of complex network services by enabling all players, not just vendors, to develop and deploy control and data plane functionality in networks. This comes at a great risk; deploying buggy code at the controller could result

in problematic flow entries at the data plane and, potentially, service disruption [13,18,48,46] and security loopholes [26,7]. Understanding and fixing such bugs is far from trivial, given the distributed and concurrent nature of computer networks and the complexity of the control plane [43].

With the advent of SDN, a large body of research on verifying network properties has emerged [32]. Static network analysis approaches [33,30,50,2,44,11] can only verify network properties on a given fixed network configuration but this may be changing very quickly (e.g. as in [1]). Another key limitation is the fact that they cannot reason about the controller program, which, itself, is responsible for the changes in the network configuration. Dynamic approaches, such as [31,39,49,23,29,47], are able to reason about network properties as changes happen (i.e. as flow entries in switches' flow tables are being added and deleted), but they cannot reason about the controller program either. As a result, when a property violation is detected, there is no straightforward way to fix the bug in the controller code, as these systems are oblivious of the running code. Identifying bugs in large and complex deployments can be extremely challenging.

Formal verification methods that include the controller code in the model of the network can solve this important problem. Symbolic execution methods, such as [45,8,11,28,14,5,12], evaluate programs using symbolic variables accumulating path-conditions along the way that then can be solved logically. However, they suffer from the path explosion problem caused by loops and function calls which means verification does not scale to larger controller programs (bug finding still works but is limited). Model checking SDNs is a promising area even though only few studies have been undertaken [28,3,8,42,34,35]. Networks and controller can be naturally modelled as transition systems. State explosion is always a problem but can be mitigated by using abstraction and optimisation techniques (i.e. partial order reductions). At the same time, modern model checkers [21,6,9,25,20] are very efficient.

NetSMC [28] uses a bespoke *symbolic* model checking algorithm for checking properties given a subset of computation tree logic that allows quantification only over all paths. As a result, this approach scales relatively well, but the requirement that only one packet can travel through the network at any time is very restrictive and ignores race conditions. NICE [8] employs model checking but only looks at a limited amount of input packets that are extracted through symbolically executing the controller code. As a result, it is a bug-finding tool only. The authors in [42] propose a model checking approach that can deal with dynamic controller updates and an arbitrary number of packets but require manually inserted non-interference lemmas that constrain the set of packets that can appear in the network. This significantly limits its applicability in realistic network deployments. Kuai [34] overcomes this limitation by introducing model-specific partial order reductions (PORs) that result in pruning the state space by avoiding redundant explorations. However, it has limitations explained at the end of this section.

In this paper, we take a step further towards the full realisation of model checking real-world SDNs by introducing MOCS (MOdel Checking for Software

defined networks)[1], a highly expressive, optimised SDN model which we implemented in UPPAAL[2] [6]. MOCS, compared to the state of the art in model checking SDNs, can model network behaviour more realistically and verify larger deployments using fewer resources. The main contributions of this paper are:
Model Generality. The proposed network model is closer to the OpenFlow standard than previous models (e.g. [34]) to reflect commonly exhibited behaviour between the controller and network switches. More specifically, it allows for race conditions between control messages and includes a significant number of OpenFlow interactions, including barrier response messages. In our experimentation section, we present families of elusive bugs that can be efficiently captured by MOCS.
Model Checking Optimisations. To tackle the state explosion problem we propose context-dependent *partial order reductions* by considering the concrete control program and specification in question. We establish the soundness of the proposed optimisations. Moreover, we propose *state representation optimisations*, namely packet and rule indexing, identification of packet equivalence classes and bit packing, to improve performance. We evaluate the benefits from all proposed optimisations in §4.

Our model has been inspired by Kuai [34]. According to the contributions above, however, we consider MOCS to be a considerable improvement. We model more OpenFlow messages and interactions, enabling us to check for bugs that [34] cannot even express (see discussion in §4.2). Our context-dependent PORs systematically explore possibilities for optimisation. Our optimisation techniques still allow MOCS to run at least as efficiently as Kuai, often with even better performance.

2 Software-Defined Network Model

A key objective of our work is to enable the verification of network-wide properties in real-world SDNs. In order to fulfil this ambition, we present an extended network model to capture complex interactions between the SDN controller and the network. Below we describe the adopted network model, its state and transitions.

2.1 Formal Model Definition

The formal definition of the proposed SDN model is by means of an action-deterministic transition system. We parameterise the model by the underlying network topology λ and the controller program CP in use, as explained further below (§2.2).

[1] A release of MOCS is publicly available at `https://tinyurl.com/y95qtv5k`
[2] UPPAAL has been chosen as future plans include extending the model to timed actions like e.g. timeouts. Note that the model can be implemented in any model checker.

Definition 1. *An SDN model is a 6-tuple* $\mathcal{M}_{(\lambda,\text{CP})} = (S, s_0, A, \hookrightarrow, AP, L)$*, where* S *is the set of all states the SDN may enter,* s_0 *the initial state,* A *the set of actions which encode the events the network may engage in,* $\hookrightarrow \subseteq S \times A \times S$ *the transition relation describing which execution steps the system undergoes as it perform actions,* AP *a set of atomic propositions describing relevant state properties, and* $L : S \to 2^{AP}$ *is a labelling function, which relates to any state* $s \in S$ *a set* $L(s) \in 2^{AP}$ *of those atomic propositions that are true for* s*. Such an SDN model is composed of several smaller systems, which model network components (hosts, switches and the controller) that communicate via queues and, combined, give rise to the definition of* $\hookrightarrow$*. The states of an SDN transition system are 3-tuples* (π, δ, γ)*, where* π *represents the state of each host,* δ *the state of each switch, and* γ *the controller state. The components are explained in §2.2 and the transitions* $\hookrightarrow$ *in §2.3.*

Figure 1 illustrates a high-level view of OpenFlow interactions (left side), modelled actions and queues (right side).

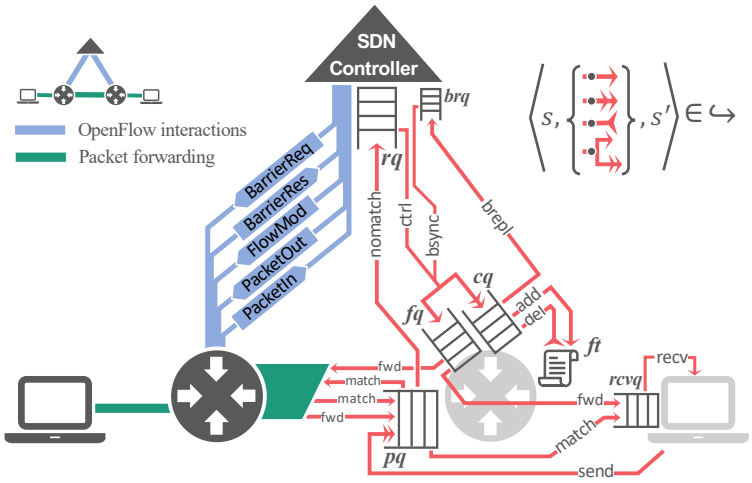


Fig. 1: A high-level view of OpenFlow interactions using OpenFlow specification terminology (left half) and the modelled actions (right half). A red solid-line arrow depicts an action which, when fired, (1) dequeues an item from the queue the arrow begins at, and (2) adds an item in the queue the arrowhead points to (or multiple items if the arrow is double-headed). Deleting an item from the target queue is denoted by a reverse arrowhead. A forked arrow denotes multiple targeted queues.

2.2 SDN Model Components

Throughout we will use the common 'dot-notation' ($\_.\_$) to refer to components of composite gadgets (tuples), e.g. queues of switches, or parts of the state. We use obvious names for the projections functions like $s.\delta.sw.pq$ for the packet queue of the switch sw in state s. At times we will also use t_1 and t_2 for the first and second projection of tuple t.

Network Topology. A location (n, pt) is a pair of a node (host or switch) n and a port pt. We describe the network topology as a bijective function $\lambda : (Switches \cup Hosts) \times Ports \to (Switches \cup Hosts) \times Ports$ consisting of a set of directed edges $\langle (n, pt), (n', pt') \rangle$, where pt' is the input port of the switch or host n' that is connected to port pt at host or switch n. *Hosts*, *Switches* and *Ports* are the (finite) sets of all hosts, switches and ports in the network, respectively. The topology function is used when a packet needs to be forwarded in the network. The location of the next hop node is decided when a *send*, *match* or *fwd* action (all defined further below) is fired. Every SDN model is w.r.t. a fixed topology λ that does not change.

Packets. Packets are modelled as finite bit vectors and transferred in the network by being stored to the queues of the various network components. A $packet \in Packets$ (the set of all packets that can appear in the network) contains bits describing the proof-relevant header information and its location *loc*.

Hosts. Each $host \in Hosts$, has a packet queue (*rcvq*) and a finite set of ports which are connected to ports of other switches. A host can send a packet to one or more switches it is connected to (*send* action in Figure 1) or receive a packet from its own *rcvq* (*recv* action in Figure 1). Sending occurs repeatedly in a non-deterministic fashion which we model implicitly via the $(0, \infty)$ abstraction at switches' packet queues, as discussed further below.

Switches. Each $switch \in Switches$, has a flow table (*ft*), a packet queue (*pq*), a control queue (*cq*), a forwarding queue (*fq*) and one or more ports, through which it is connected to other switches and/or hosts. A flow table $ft \subseteq Rules$ is a set of forwarding rules (with *Rules* being the set of all rules). Each one consists of a tuple $(priority, pattern, ports)$, where $priority \in \mathbb{N}$ determines the priority of the rule over others, *pattern* is a proposition over the proof-relevant header of a packet, and *ports* is a subset of the switch's ports. Switches match packets in their packet queues against rules (i.e. their respective *pattern*) in their flow table (*match* action in Figure 1) and forward packets to a connected device (or final destination), accordingly. Packets that cannot be matched to any rule are sent to the controller's request queue (*rq*) (*nomatch* action in Figure 1); in OpenFlow, this is done by sending a *PacketIn* message. The forwarding queue *fq* stores packets forwarded by the controller in *PacketOut* messages. The control queue stores messages sent by the controller in *FlowMod* and *BarrierReq* messages. *FlowMod* messages contain instructions to add or delete rules from the flow table (that trigger *add* and *del* actions in Figure 1). *BarrierReq* messages contain barriers to synchronise the addition and removal of rules. MOCS conforms to the OpenFlow specifications and always execute instructions in an interleaved fashion obeying the ordering constraints imposed by barriers.

OpenFlow Controller. The controller is modelled as a finite state automaton embedded into the overall transition system. A controller program CP, as used to parametrise an SDN model, consists of $(CS, pktIn, barrierIn)$. It uses its own local state $cs \in CS$, where CS is the finite set of control program states. Incoming *PacketIn* and *BarrierRes* messages from the SDN model are stored in separate queues (*rq* and *brq*, respectively) and trigger *ctrl* or *bsync* actions (see Figure 1) which are then processed by the controller program in its current state. The controller's corresponding handler, *pktIn* for *PacketIn* messages and *barrierIn* for *BarrierRes* messages, responds by potentially changing its local state and sending messages to a subset of *Switches*, as follows. A number of *PacketOut* messages – pairs of (*pkt, ports*) – can be sent to a subset of *Switches*. Such a message is stored in a switch's forward queue and instructs it to forward packet *pkt* along the ports *ports*. The controller may also send any number of *FlowMod* and *BarrierReq* messages to the control queue of any subset of *Switches*. A *FlowMod* message may contain an *add* or *delete* rule modification instruction. These are executed in an arbitrary order by switches, and *barriers* are used to synchronise their execution. Barriers are sent by the controller in *BarrierReq* messages. OpenFlow requires that a response message (*BarrierRes*) is sent to the controller by a switch when a barrier is consumed from its control queue so that the controller can synchronise subsequent actions. Our model includes a *brepl* action that models the sending of a *BarrierRes* message from a switch to the controller's barrier reply queue (*brq*), and a *bsync* action that enables the controller program to react to barrier responses.

Queues. All queues in the network are modelled as *finite* state. Packet queues *pq* for switches are modelled as multisets, and we adopt $(0, \infty)$ abstraction [40]; i.e. a packet is assumed to appear either zero or an arbitrary (unbounded) amount of times in the respective multiset. This means that once a packet has arrived at a switch or host, (infinitely) many other packets of the same kind repeatedly arrive at this switch or host. Switches' forwarding queues *fq* are, by contrast, modelled as sets, therefore if multiple identical packets are sent by the controller to a switch, only one will be stored in the queue and eventually forwarded by the switch. The controller's request *rq* and barrier reply queues *brq* are modelled as sets as well. Hosts' receive queues *rcvq* are also modelled as sets. Controller queues *cq* at switches are modelled as a finite sequence of sets of control messages (representing add and remove rule instructions), interleaved by any number of barriers. As the number of barriers that can appear at any execution is finite, this sequence is finite.

2.3 Guarded Transitions

Here we provide a detailed breakdown of the transition relation $s \xrightarrow{\alpha(\vec{a})} s'$ for each action $\alpha(\vec{a}) \in A(s)$, where $A(s)$ the set of all enabled actions in s in the proposed model (see Figure 1). Transitions are labelled by action names α with arguments $\vec{a}$. The transitions are only enabled in state s if s satisfies certain conditions called *guards* that can refer to the arguments $\vec{a}$. In guards, we make use of predicate $bestmatch(sw, r, pkt)$ that expresses that r is the highest priority

rule in $sw.ft$ that matches pkt's header. Below we list all possible actions with their respective guards.

$\boldsymbol{send(h, pt, pkt)}$. Guard: $true$. This transition models packets arriving in the network in a non-deterministic fashion. When it is executed, pkt is added to the packet queue of the network switch connected to the port pt of host h (or, formally, to $\lambda(h, pt)_1.pq$, where λ is the topology function described above). As described in §3.2, only relevant representatives of packets are actually sent by end-hosts. This transition is unguarded, therefore it is always enabled.

$\boldsymbol{recv(h, pkt)}$. Guard: $pkt \in h.rcvq$. This transition models hosts receiving (and removing) packets from the network and is enabled if pkt is in h's receive queue.

$\boldsymbol{match(sw, pkt, r)}$. Guard: $pkt \in sw.pq \wedge r \in sw.ft \wedge bestmatch(sw, r, pkt)$. This transition models matching and forwarding packet pkt to zero or more next hop nodes (hosts and switches), as a result of highest priority matching of rule r with pkt. The packet is then copied to the packet queues of the connected hosts and/or switches, by applying the topology function to the port numbers in the matched rule; i.e. $\lambda(sw, pt)_1.pq, \forall pt \in r.ports$. Dropping packets is modelled by having a special 'drop' port that can be included in rules. The location of the forwarded packet(s) is updated with the respective destination (switch/host, port) pair; i.e. $\lambda(sw, pt)$. Due to the $(0, \infty)$ abstraction, the packet is not removed from $sw.pq$.

$\boldsymbol{nomatch(sw, pkt)}$. Guard: $pkt \in sw.pq \wedge \nexists r \in sw.ft\ .\ bestmatch(sw, r, pkt)$. This transition models forwarding a packet to the OpenFlow controller when a switch does not have a rule in its forwarding table that can be matched against the packet header. In this case, pkt is added to rq for processing. pkt is not removed from $sw.pq$ due to the supported $(0, \infty)$ abstraction.

$\boldsymbol{ctrl(pkt, cs)}$. Guard: $pkt \in controller.rq$. This transition models the execution of the packet handler by the controller when packet pkt, that was previously sent by switch $pkt.loc_1$, is available in rq. The controller's packet handler function $pktIn(pkt.loc_1, pkt, cs)$ is executed which, in turn (i) reads the current controller state cs and changes it according to the controller program, (ii) adds a number of rules, interleaved with any number of barriers, into the cq of zero or more switches, and (iii) adds zero or more forwarding messages, each one including a packet along with a set of ports, to the fq of zero or more switches.

$\boldsymbol{fwd(sw, pkt, ports)}$. Guard: $(pkt, ports) \in sw.fq$. This transition models forwarding packet pkt that was previously sent by the controller to sw's forwarding queue $sw.fq$. In this case, pkt is removed from $sw.fq$ (which is modelled as a set), and added to the pq of a number of network nodes (switches and/or hosts), as defined by the topology function $\lambda(sw, pt)_1.pq, \forall pt \in ports$. The location of the forwarded packet(s) is updated with the respective destination (switch/host, port) pair; i.e. $\lambda(n, pt)$.

$\boldsymbol{FM(sw, r)}$, where $FM \in \{add, del\}$. Guard: $(FM, r) \in head(sw.cq)$. These transitions model the addition and deletion, respectively, of a rule in the flow table of switch sw. They are enabled when one or more add and del control messages are in the set at the head of the switch's control queue. In this case, r is added to – or deleted from, respectively – $sw.ft$ and the control message is deleted from the set at the head of cq. If the set at the head of cq becomes empty it is removed.

If then the next item in *cq* is a barrier, a *brepl* transition becomes enabled (see below).

***brepl*(*sw*, *xid*)**. Guard: $b(xid) = head(sw.cq)$. This transition models a switch sending a barrier response message, upon consuming a barrier from the head of its control queue; i.e. if $b(xid)$ is the head of $sw.cq$, where $xid \in \mathbb{N}$ is an identifier for the barrier set by the controller, $b(xid)$ is removed and the barrier reply message $br(sw, xid)$ is added to the controller's *brq*.

***bsync*(*sw*, *xid*, *cs*)**. Guard: $br(sw, xid) \in controller.brq$. This transition models the execution of the barrier response handler by the controller when a barrier response sent by switch *sw* is available in *brq*. In this case, $br(sw, xid)$ is removed from the *brq*, and the controller's barrier handler $barrierIn(sw, xid, cs)$ is executed which, in turn (i) reads the current controller state *cs* and changes it according to the controller program, (ii) adds a number of rules, interleaved with any number of barriers, into the *cq* of zero or more switches, and (iii) adds zero or more forwarding messages, each one including a packet along with a set of ports, to the *fq* of zero or more switches.

An example run. In Figure 2, we illustrate a sequence of MOCS transitions through a simple packet forwarding example. The run starts with a *send* transition; packet p is copied to the packet queue of the switch in black. Initially, switches' flow tables are empty, therefore p is copied to the controller's request queue (*nomatch* transition); note that p remains in the packet queue of the switch in black due to the $(0, \infty)$ abstraction. The controller's packet handler is then called (*ctrl* transition) and, as a result, (1) p is copied to the forwarding queue of the switch in black, (2) rule r_1 is copied to the control queue of the switch in black, and (3) rule r_2 is copied to the control queue of the switch in white. Then, the switch in black forwards p to the packet queue of the switch in white (*fwd* transition). The switch in white installs r_2 in its flow table (*add* transition) and then matches p with the newly installed rule and forwards it to the receive queue of the host in white (*match* transition), which removes it from the network (*recv* transition).

2.4 Specification Language

In order to specify properties of packet flow in the network, we use LTL formulas without "next-step" operator $\bigcirc$[3], where atomic formulae denoting properties of states of the transition system, i.e. SDN network. In the case of safety properties, i.e. an invariant w.r.t. states, the $\text{LTL}_{\setminus\{\bigcirc\}}$ formula is of the form $\Box\varphi$, i.e. has only an outermost $\Box$ temporal connective.

Let P denote unary predicates on packets which encode a property of a packet based on its fields. An atomic *state condition* (proposition) in AP is either of the following: (i) existence of a packet *pkt* located in a packet queue (*pq*) of a switch or in a receive queue (*rcvq*) of a host that satisfies P (we denote this by

[3] This is the largest set of formulae supporting the partial order reductions used in §3, as stutter equivalence does not preserve the truth value of formulae with the $\bigcirc$.

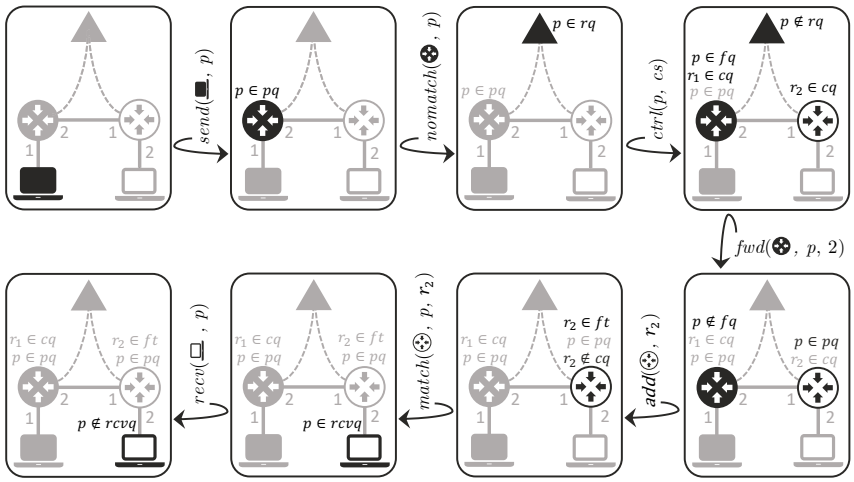


Fig. 2: Forwarding p from 💻 to 💻. Non greyed-out icons are the ones whose state changes in the current transition.

$\exists pkt{\in}n.pq\,.\,P(pkt)$ with $n \in \mathit{Switches}$, and $\exists pkt{\in}h.rcvq\,.\,P(pkt)$ with $h \in \mathit{Hosts}$)[4]; (ii) the controller is in a specific *controller* state $q \in CS$, denoted by a unary predicate symbol $Q(q)$ which holds in system state $s \in S$ if $q = s.\gamma.cs$. The specification logic comprises first-order formula with equality on the finite domains of switches, hosts, rule priorities, and ports which are *state-independent* (and decidable).

For example, $\exists pkt{\in}sw.pq\,.\,P(pkt)$ represents the fact that the packet predicate $P(\_)$ is true for at least one packet pkt in the pq of switch sw. For every atomic packet proposition $P(pkt)$, also its negation $\neg P(pkt)$ is an atomic proposition for the reason of simplifying syntactic checks of formulae in Table 1 in the next section. Note that universal quantification over packets in a queue is a derived notion. For instance, $\forall pkt{\in}n.pq\,.\,P(pkt)$ can be expressed as $\nexists pk{\in}n.pq\,.\,\neg P(pkt)$. Universal and existential quantification over switches or hosts can be expressed by finite iterations of $\wedge$ and $\vee$, respectively.

In order to be able to express that a condition holds when a certain event happened, we add to our propositions instances of *propositional dynamic logic* [41,17]. Given an action $\alpha(\cdot) \in A$ and a proposition P that may refer to any variables in $\vec{x}$, $[\alpha(\vec{x})]P$ is also a proposition and $[\alpha(\vec{x})]P$ is true if, and only if, after firing transition $\alpha(\vec{a})$ (to get to the current state), P holds with the variables in $\vec{x}$ bound to the corresponding values in the actual arguments $\vec{a}$. With the help of those basic modalities one can then also specify that more complex events occurred. For instance, dropping of a packet due to a *match* or *fwd* action can

[4] Note that these are *atomic* propositions despite the use of the existential quantifier notation.

be expressed by $[match(sw, pkt, r)](r.fwd\_port = \texttt{drop}) \wedge [fwd(sw, pkt, pt)](pt = \texttt{drop})$. Such predicates derived from modalities are used in §B-CP5.

The meaning of temporal LTL operators is standard depending on the trace of a transition sequence $s_0 \xhookrightarrow{\alpha_1} s_1 \xhookrightarrow{\alpha_2} \ldots$. The trace $L(s_0)L(s_1)\ldots L(s_i)\ldots$ is defined as usual. For instance, trace $L(s_0)L(s_1)L(s_2)\ldots$ satisfies invariant $\Box\varphi$ if each $L(s_i)$ implies φ.

3 Model Checking

In order to verify desired properties of an SDN, we use its model as described in Def. 1 and apply model checking. In the following we propose optimisations that significantly improve the performance of model checking.

3.1 Contextual Partial-Order Reduction

Partial order reduction (POR) [38] reduces the number of interleavings (traces) one has to check. Here is a reminder of the main result (see [4]) where we use a stronger condition than the regular (*C4*) to deal with cycles:

Theorem 1 (Correctness of POR). *Given a finite transition system $\mathcal{M} = (S, A, \hookrightarrow, s_0, AP, L)$ that is action-deterministic and without terminal states, let $A(s)$ denote the set of actions in A enabled in state $s \in S$. Let $ample(s) \subseteq A(s)$ be a set of actions for a state $s \in S$ that satisfies the following conditions:*

C1 (Non)emptiness condition: $\varnothing \neq ample(s) \subseteq A(s)$.

C2 Dependency condition: Let $s \xhookrightarrow{\alpha_1} s_1 \ldots \xhookrightarrow{\alpha_n} s_n \xhookrightarrow{\beta} t$ be a run in $\mathcal{M}$. If $\beta \in A \backslash ample(s)$ depends on $ample(s)$, then $\alpha_i \in ample(s)$ for some $0 < i \leqslant n$, which means that in every path fragment of $\mathcal{M}$, β cannot appear before some transition from $ample(s)$ is executed.

C3 Invisibility condition: If $ample(s) \neq A(s)$ (i.e., state s is not fully expanded), then every $\alpha \in ample(s)$ is invisible.

C4 Every cycle in $\mathcal{M}^{ample}$ contains a state s such that $ample(s) = A(s)$.

where $\mathcal{M}^{ample} = (S_a, A, \hookrightarrow\!\!\!\!\rightarrow, s_0, AP, L_a)$ is the new, optimised, model defined as follows: let $S_a \subseteq S$ be the set of states reachable from the initial state s_0 under $\hookrightarrow\!\!\!\!\rightarrow$, let $L_a(s) = L(s)$ for all $s \in S_a$ and define $\hookrightarrow\!\!\!\!\rightarrow \subseteq S_a \times A \times S_a$ inductively by the rule

$$\frac{s \xhookrightarrow{\alpha} s'}{s \xhookrightarrow{\alpha} s'} \qquad \text{if } \alpha \in ample(s)$$

If $ample(s)$ satisfies conditions (C1)-(C4) as outlined above, then for each path in $\mathcal{M}$ there exists a stutter-trace equivalent path in $\mathcal{M}^{ample}$, and vice-versa, denoted $\mathcal{M} \overset{\text{st}}{\equiv} \mathcal{M}^{ample}$.

The intuitive reason for this theorem to hold is the following: Assume an action sequence $\alpha_i \ldots \alpha_{i+n}\beta$ that reaches the state s, and β is *independent* of $\{\alpha_i, \ldots \alpha_{i+n}\}$. Then, one can permute β with α_{i+n} through α_i successively n times.

One can therefore construct the sequence $\beta\alpha_i...\alpha_{i+n}$ that also reaches the state s. If this shift of β does not affect the labelling of the states with atomic propositions (β is called *invisible* in this case), then it is not detectable by the property to be shown and the permuted and the original sequence are equivalent w.r.t. the property and thus don't have to be checked both. One must, however, ensure, that in case of loops (infinite execution traces) the ample sets do not *preclude* some actions to be fired altogether, which is why one needs ($C4$).

The more actions that are both stutter and provably independent (also referred to as *safe actions* [22]) there are, the smaller the transition system, and the more efficient the model checking. One of our contributions is that we attempt to identify *as many safe actions as possible* to make PORs more widely applicable to our model.

The PORs in [34] consider only dependency and invisibility of *recv* and *barrier* actions, whereas we explore systematically all possibilities for applications of Theorem 1 to reduce the search space. When identifying safe actions, we consider (1) the actual controller program CP, (2) the topology λ and (3) the state formula φ to be shown invariant, which we call the *context* CTX of actions. It turns out that two actions may be dependent in a given context of abstraction while independent in another context, and similarly for invisibility, and we exploit this fact. The argument of the action thus becomes relevant as well.

Definition 2 (Safe Actions). *Given a context* $\text{CTX} = (\text{CP}, \lambda, \varphi)$*, and SDN model* $\mathcal{M}_{(\lambda,\text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$*, an action* $\alpha(\cdot) \in A(s)$ *is called 'safe' if it is independent of any other action in* A *and invisible for* φ*. We write safe actions* $\check{\alpha}(\cdot)$*.*

Definition 3 (Order-sensitive Controller Program). *A controller program* CP *is order-sensitive if there exists a state* $s \in S$ *and two actions* α, β *in* $\{ctrl(\cdot), bsync(\cdot)\}$ *such that* $\alpha, \beta \in A(s)$ *and* $s \overset{\alpha}{\hookrightarrow} s_1 \overset{\beta}{\hookrightarrow} s_2$ *and* $s \overset{\beta}{\hookrightarrow} s_3 \overset{\alpha}{\hookrightarrow} s_4$ *with* $s_2 \neq s_4$*.*

Definition 4. *Let* φ *be a state formula. An action* $\alpha \in A$ *is called '*φ*-invariant' if* $s \models \varphi$ *iff* $\alpha(s) \models \varphi$ *for all* $s \in S$ *with* $\alpha \in A(s)$*.*

Lemma 1. *For transition system* $\mathcal{M}_{(\lambda,\text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$ *and a formula* $\varphi \in LTL_{\setminus\{\bigcirc\}}$*,* $\alpha \in A$ *is safe* iff $\bigwedge_{i=1}^{3} Safe_i(\alpha)$*, where* $Safe_i$*, given in Table 1, are per-row.*

Proof. See Appendix A. □

Theorem 2 (POR instance for SDN). *Let* $(\text{CP}, \lambda, \varphi)$ *be a context such that* $\mathcal{M}_{(\lambda,\text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$ *is an SDN network model from Def. 1; and let safe actions be as in Def. 2. Further, let* $ample(s)$ *be defined by:*

$$ample(s) = \begin{cases} \{\alpha \in A(s) \mid \alpha \text{ safe } \} & \text{if } \{\alpha \in A(s) \mid \alpha \text{ safe } \} \neq \varnothing \\ A(s) & \text{otherwise} \end{cases}$$

Then, ample satisfies the criteria of Theorem 1 and thus $\mathcal{M}_{(\lambda,\text{CP})} \overset{\text{st}}{\equiv} \mathcal{M}^{ample}_{(\lambda,\text{CP})}$ [5]

[5] Stutter equivalence here implicitly is defined w.r.t. the atomic propositions appearing in φ, but this suffices as we are just interested in the validity of φ.

Table 1: Safeness Predicates

Action $Safe_1(\alpha)$	**Independence** $Safe_2(\alpha)$	**Invisibility** $Safe_3(\alpha)$
$\alpha = ctrl(pk, cs)$	CP is not order-sensitive	`if` $Q(q)$ occurs in φ, where $q \in CS$, `then` α is φ-invariant.
$\alpha = bsync(sw, xid, cs)$	CP is not order-sensitive	`if` $Q(q)$ occurs in φ, where $q \in CS$, `then` α is φ-invariant.
$\alpha = fwd(sw, pk, ports)$	$\top$	`if` $\exists pk{\in}b.q\,.\,P(pk)$ occurs in φ, for any $b \in \{sw\} \cup \{\lambda(sw,p)_1 \mid p \in ports\}$ and $q \in \{pq, recvq\}$, `then` α is φ-invariant.
$\alpha = brepl(sw, xid)$	$\top$	$\top$
$\alpha = recv(h, pk)$	$\top$	`if` $\exists pk{\in}h.rcvq\,.\,P(pk)$ occurs in φ, `then` α is φ-invariant.

Proof.

C1 The (non)emptiness condition is trivial since by definition of $ample(s)$ it follows that $ample(s) = \varnothing$ iff $A(s) = \varnothing$.

C2 By assumption $\beta \in A \backslash ample(s)$ depends on $ample(s)$. But with our definition of $ample(s)$ this is impossible as all actions in $ample(s)$ are safe and by definition independent of all other actions.

C3 The validity of the invisibility condition is by definition of *ample* and safe actions.

C4 We now show that every cycle in $\mathcal{M}^{ample}_{(\lambda,\text{CP})}$ contains a fully expanded state s, i.e. a state s such that $ample(s) = A(s)$. By definition of $ample(s)$ in Thm. 2 it is equivalent to show that there is no cycle in $\mathcal{M}^{ample}_{(\lambda,\text{CP})}$ consisting of safe actions only. We show this by contradiction, assuming such a cycle of only safe actions exists. There are five safe action types to consider: *ctrl*, *fwd*, *brepl*, *bsync* and *recv*. Distinguish two cases.

Case 1. A sequence of safe actions of same type. Let us consider the different safe actions:

- Let ρ an execution of $\mathcal{M}^{ample}_{(\lambda,\text{CP})}$ which consists of only one type of *ctrl*-actions:

$$\rho = s_1 \xrightarrow{ctrl(pkt_1, cs_1)} s_2 \xrightarrow{ctrl(pkt_2, cs_2)} \ldots s_{i-1} \xrightarrow{ctrl(pkt_{i-1}, cs_{i-1})} s_i$$

  Suppose ρ is a cycle. According to the *ctrl* semantics, for each transition $s \xrightarrow{ctrl(pkt,cs)} s'$, where $s = (\pi, \delta, \gamma)$, $s' = (\pi', \delta', \gamma')$, it holds that $\gamma'.rq = \gamma.rq \backslash \{pkt\}$ as we use sets to represent *rq* buffers. Hence, for the execution ρ it holds $\gamma_i.rq = \gamma_1.rq \backslash \{pkt_1, pkt_2, \ldots pkt_{i-1}\}$ which implies that $s_1 \neq s_i$. Contradiction.
- Let ρ an execution which consists of only one type of *fwd*-actions: similar argument as above since *fq*-s are represented by sets and thus forward messages are removed from *fq*.
- Let ρ an execution which consists of only one type of *brepl*-actions: similar argument as above since control messages are removed from *cq*.

- Let ρ an execution which consists of only one type of *bsync*-actions: similar argument as above, as barrier reply messages are removed from *brq*-s that are represented by sets.
- Let ρ an execution which consists of only one type of *recv*-actions: similar argument as above, as packets are removed from *rcvq* buffers that are represented by sets.

Case 2. A sequence of different safe actions. Suppose there exists a cycle with mixed safe actions starting in s_1 and ending in s_i. Distinguish the following cases.

i) There exists at least a *ctrl* and/or a *bsync* action in the cycle. According to the effects of safe transitions, the *ctrl* action will change to a state with smaller *rq* and the *bsync* will always switch to a state with smaller *brq*. It is important here that *ctrl* does not interfere with *bsync* regarding *rq, brq*, and no safe action of other type than *ctrl* and *bsync* accesses *rq* or *brq*. This implies that $s_1 \neq s_i$. Contradiction.

ii) Neither *ctrl*, nor *bsync* actions in the cycle.
   a) There is a *fwd* and/or *brepl* in the cycle: *fwd* will always switch to a state with smaller *fq* and *brepl* will always switch to a state with smaller *cq* (*brepl* and *recv* do not interfere with *fwd*). This implies that $s_1 \neq s_i$. Contradiction.
   b) There is neither *fwd* nor *brepl* in the cycle. This means that only *recv* is in the cycle which is already covered by the first case.

□

Due to the definition of the transition system via ample sets, each safe action is immediately executed after its enabling one. Therefore, one can merge every transition of a safe action with its precursory enabling one. Intuitively, the semantics of the merged action is defined as the successive execution of its constituent actions. This process can be repeated if there is a chain of safe actions; for instance, in the case of $s \xrightarrow{nomatch(sw,pkt)} s' \xrightarrow{ctrl(pkt,cs)} s'' \xrightarrow{fwd(sw,pkt,ports)} s'''$ where each transition enables the next and the last two are assumed to be safe. These transitions can be merged into one, yielding a stutter equivalent trace as the intermediate states are invisible (w.r.t. the context and thus the property to be shown) by definition of safe actions.

3.2 State Representation

Efficient state representation is crucial for minimising MOCS's memory footprint and enabling it to scale up to relatively large network setups.

Packet and Rule Indexing. In MOCS, only a single instance of each packet and rule that can appear in the modelled network is kept in memory. An index is then used to associate queues and flow tables with packets and rules, with a single bit indicating their presence (or absence). This data structure is illustrated in Figure 3. For a data packet, a value of 1 in the *pq* section of the entry indicates that infinite copies of it are stored in the packet queue of the respective switch. A value of 1 in the *fq* section indicates that a single copy of the packet is stored in

the forward queue of the respective switch. A value of 1 in the *rq* section indicates that a copy of the packet sent by the respective switch (when a *nomatch* transition is fired) is stored in the controller's request queue. For a rule, a value of 1 in the *ft* section indicates that the rule is installed in the respective switch's flow table. A value of 1 in the *cq* section indicates that the rule is part of a *FlowMod* message in the respective switch's control queue.

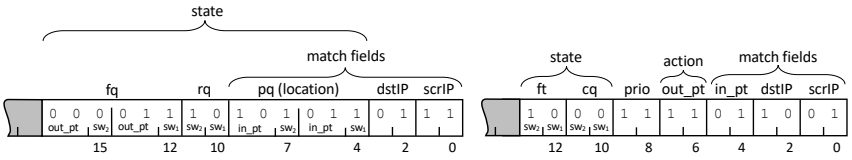


Fig. 3: Packet (left) and rule (right) indices

The proposed optimisation enables scaling up the network topology by minimising the required memory footprint. For every switch, MOCS only requires a few bits in each packet and rule entry in the index.

Discovering equivalence classes of packets. Model checking with all possible packets including all specified fields in the OpenFlow standard would entail a huge state space that would render any approach unusable. Here, we propose the discovery of equivalence classes of packets that are then used for model checking. We first remove all fields that are not referenced in a statement or rule creation or deletion in the controller program. Then, we identify packet classes that would result in the same controller behaviour. Currently, as with the rest of literature, we focus on simple controller programs where such equivalence classes can be easily identified by analysing static constraints and rule manipulation in the controller program. We then generate one representative packet from each class and assign it to all network switches that are directly connected to end-hosts; i.e. modelling clients that can send an arbitrarily large number of packets in a non-deterministic fashion. We use the minimum possible number of bits to represent the identified equivalence classes. For example, if the controller program exerts different behaviour if the destination TCP port of a packet is 22 (i.e. destined to an SSH server) or not, we only use a 1-bit field to model this behaviour.

Bit packing. We reduce the size of each recorded state by employing bit packing using the `int32` type supported by UPPAAL, and bit-level operations for the entries in the packet and rule indices, as well as for the packets and rules themselves.

4 Experimental Evaluation

In this section, we experimentally evaluate MOCS by comparing it with the state of the art, in terms of performance (verification throughput and memory footprint) and model expressivity. We have implemented MOCS in UPPAAL [6] as a network of parallel automata for the controller and network switches, which communicate asynchronously by writing/reading packets to/from queues that

are part of the model discussed in §2. As discussed in §3, this is implemented by directly manipulating the packet and rule indices.

Throughout this section we will be using three examples of network controllers: (1) A *stateless firewall* (§B-CP1) requires the controller to install rules to network switches that enable them to decide whether to forward a packet towards its destination or not; this is done in a stateless fashion, i.e. without having to consider any previously seen packets. For example, a controller could configure switches to block all packets whose destination TCP port is SSH. (2) A *stateful firewall* (§B-CP2) is similar to the stateless one but decisions can take into account previously seen packets. A classic example of this is to allow bi-directional communication between two end-hosts, when one host opens a TCP connection to the other. Then, traffic flowing from the other host back to the connection initiator should be allowed to go through the switches on the reverse path. (3) A *MAC learning application* (§B-CP3) enables the controller and switches to learn how to forward packets to their destinations (identified with respective MAC addresses). A switch sends a *PacketIn* message to the controller when it receives a packet that it does not know how to forward. By looking at this packet, the controller learns a mapping of a source switch (or host) to a port of the requesting switch. It then installs a rule (by sending a *FlowMod* message) that will allow that switch to forward packets back to the source switch (or host), and asks the requesting switch (by sending a *PacketOut* message) to flood the packet to all its ports except the one it received the packet from. This way, the controller eventually learns all mappings, and network switches receive rules that enable them to forward traffic to their neighbours for all destinations in the network.

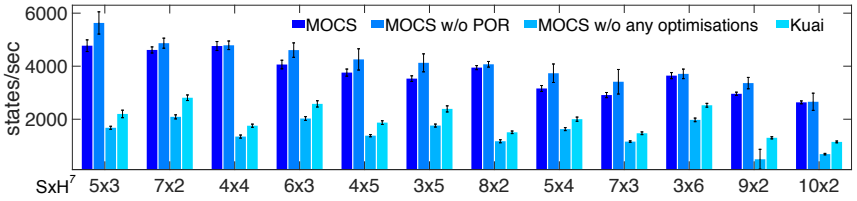
4.1 Performance Comparison

We measure MOCS's performance, and also compare it against Kuai [34][6] using the examples described above, and we investigate the behaviour of MOCS as we scale up the network (switches and clients/servers). We report three metrics: (1) *verification throughput* in visited states per second, (2) number of visited states, and (3) required memory. We have run all verification experiments on an 18-Core iMac pro, 2.3GHz Intel Xeon W with 128GB DDR4 memory.

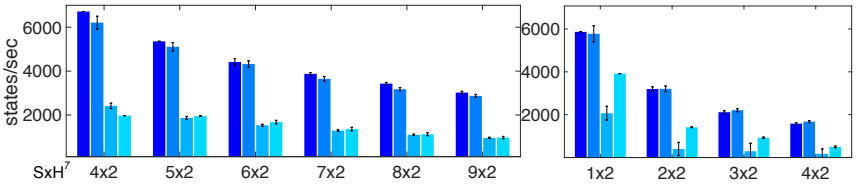
Verification throughput. We measure the verification throughput when running a single experiment at a time on one CPU core and report the average and standard deviation for the first 30 minutes of each run. In order to assess how MOCS's different optimisations affect its performance, we report results for the following system variants: (1) MOCS, (2) MOCS without POR, (3) MOCS without any optimisations (neither POR, state representation), and (4) Kuai. Figure 4 shows the measured throughput (with error bars denoting standard deviation).

For the MAC learning and stateless firewall applications, we observe that MOCS performs significantly better than Kuai for all different network setups

[6] Note that parts of Kuai's source code are not publicly available, therefore we implemented it's model in UPPAAL.



(a) MAC Learning Switch



(b) Stateless Firewall **(c)** Stateful Firewall

Fig. 4: Performance Comparison – Verification Throughput

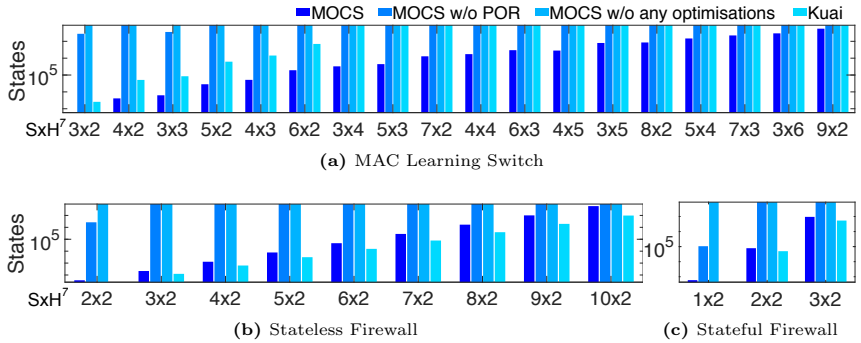
and sizes[7], achieving at least double the throughput Kuai does. The throughput performance is much better for the stateful firewall, too. This is despite the fact that, for this application, Kuai employs the unrealistic optimisation where the *barrier* transition forces the immediate update of the forwarding state. In other words, MOCS is able to explore significantly more states and identify bugs that Kuai cannot (see §4.2).

The computational overhead induced by our proposed PORs is minimal. This overhead occurs when PORs require dynamic checks through the safety predicates described in Table 1. This is shown in Figure 4a, where, in order to decide about the (in)visibility of $fwd(sw, pk, pt)$ actions, a lookup is performed in the history-array of packet pk, checking whether the bit which corresponds to switch sw', which is connected with port pt of sw, is set. On the other hand, if a POR does not require any dynamic checks, no penalty is induced, as shown in Figures 4b and 4c, where the throughput when the PORs are disabled is almost identical to the case where PORs are enabled. This is because it has been statically established at a pre-analysis stage that all actions of a particular type are always safe for any argument/state. It is important to note that even when computational overhead is induced, PORs enable MOCS to scale up to larger networks because the number of visited states can be significantly reduced, as discussed below.

In order to assess the contribution of the state representation optimisation in MOCS's performance, we measure the throughput when both PORs and state representation optimisations are disabled. It is clear that they contribute significantly to the overall throughput; without these the measured throughput was at least less than half the throughput when they were enabled.

Number of visited states and required memory. Minimising the number of visited states and required memory is crucial for scaling up verification to

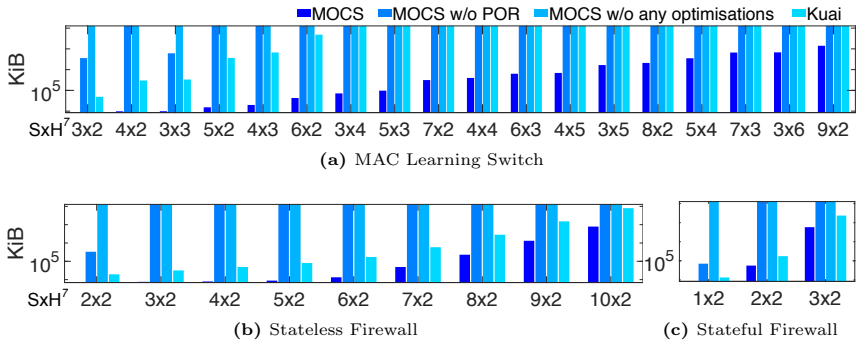
[7] S×H in Figures 4 to 6 indicates the number of switches S and hosts H.



(a) MAC Learning Switch

(b) Stateless Firewall **(c)** Stateful Firewall

Fig. 5: Performance Comparison – Visited States (logarithmic scale)



(a) MAC Learning Switch

(b) Stateless Firewall **(c)** Stateful Firewall

Fig. 6: Performance Comparison – Memory Footprint (logarithmic scale)

larger networks. The proposed partial order reductions (§3.1) and identification of packet equivalent classes aim at the former, while packet/rule indexing and bit packing aim at the latter (§3.2). In Figure 5, we present the results for the various setups and network deployments discussed above. We stopped scaling up the network deployment for each setup when the verification process required more than 24 hours or started swapping memory to disk. For these cases we killed the process and report a topped-up bar in Figures 5 and 6.

For the MAC learning application, MOCS can scale up to larger network deployments compared to Kuai, which could not verify networks consisting of more than 2 hosts and 6 switches. For that network deployment, Kuai visited ∼7m states, whereas MOCS visited only ∼193k states. At the same time, Kuai required around 48GBs of memory (7061 bytes/state) whereas MOCS needed ∼43MBs (228 bytes/state). Without the partial order reductions, MOCS can only verify tiny networks. The contribution of the proposed state representation optimisations is also crucial; in our experiments (results not shown due to lack of space), for the 6×2 network setups (the largest we could do without these

optimisations), we observed a reduction in state space (due to the identification of packet equivalence classes) and memory footprint (due to packet/rule indexing and bit packing) from ∼7m to ∼200k states and from ∼6KB per state to ∼230B per state. For the stateless and stateful firewall applications, resp., MOCS performs equally well to Kuai with respect to scaling up.

4.2 Model Expressivity

The proposed model is significantly more expressive compared to Kuai as it allows for more asynchronous concurrency. To begin with, in MOCS, controller messages sent before a barrier request message can be interleaved with all other enabled actions, other than the control messages sent after the barrier. By contrast, Kuai always flushes all control messages until the last barrier in one go, masking a large number of interleavings and, potentially, buggy behaviour. Next, in MOCS *nomatch, ctrl* and *fwd* can be interleaved with other actions. In Kuai, it is enforced a mutual exclusion concurrency control policy through the *wait*-semaphore: whenever a *nomatch* occurs the mutex is locked and it is unlocked by the *fwd* action of the thread *nomatch-ctrl-fwd* which refers to the same packet; all other threads are forced to wait. Moreover, MOCS does not impose any limit on the size of the *rq* queue, in contrast to Kuai where only one packet can exist in it. In addition, Kuai does not support notifications from the data plane to the controller for completed operations as it does not support reply messages and as a result any bug related to the fact that the controller is not synced to data-plane state changes is hidden.[8] Also, our specification language for states is more expressive than Kuai's, as we can use any property in LTL without "next", whereas Kuai only uses invariants with a single outermost $\Box$.
The MOCS extensions, however, are conservative with respect to Kuai, that is we have the following theorem (without proof, which is straightforward):

Theorem 3 (MOCS Conservativity). *Let* $\mathcal{M}_{(\lambda,\text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$ *and* $\mathcal{M}^{K}_{(\lambda,\text{CP})} = (S_K, A_K, \hookrightarrow_K, s_0, AP, L)$ *the original SDN models of MOCS and Kuai, respectively, using the same topology and controller. Furthermore, let* $\mathit{Traces}(\mathcal{M}_{(\lambda,\text{CP})})$ *and* $\mathit{Traces}(\mathcal{M}^{K}_{(\lambda,\text{CP})})$ *denote the set of all initial traces in these models, respectively. Then,* $\mathit{Traces}(\mathcal{M}^{K}_{(\lambda,\text{CP})}) \subseteq \mathit{Traces}(\mathcal{M}_{(\lambda,\text{CP})})$.

For each of the extensions mentioned above, we briefly describe an example (controller program and safety property) that expresses a bug that is impossible to occur in Kuai.
Control message reordering bug. Let us consider a stateless firewall in Figure 7a (controller is not shown), which is supposed to block incoming SSH packets from reaching the server (see §B-CP1). Formally, the safety property to be checked here is $\Box(\forall pkt \in S.rcvq\,.\,\neg pkt.\text{SSH})$. Initially, flow tables are empty. Switch A sends a *PacketIn* message to the controller when it receives the first packet from the client (as a result of a *nomatch* transition). The controller, in response

[8] There are further small extensions; for instance, in MOCS the controller can send multiple *PacketOut* messages (as OpenFlow prescribes).

to this request (and as a result of a *ctrl* transition), sends the following *FlowMod* messages to switch A; rule `r1` has the highest priority and drops all SSH packets, rule `r2` sends all packets from port 1 to port 2, and rule `r3` sends all packets from port 2 to port 1. If the packet that triggered the transition above is an SSH one, the controller drops it, otherwise, it instructs (through a *PacketOut* message) A to forward the packet to S. A bug-free controller should ensure that `r1` is installed before any other rule, therefore it must send a barrier request after the *FlowMod* message that contains `r1`. If, by mistake, the *FlowMod* message for `r2` is sent before the barrier request, A may install `r2` before `r1`, which will result in violating the given property. MOCS is able to capture this buggy behaviour as its semantics allows control messages prior to the barrier to be processed in a interleaved manner.

Fig. 7: Two networks with (a) two switches, and (b) n stateful firewall replicas

Wrong nesting level bug. Consider a correct controller program that enforces that server S (Figure 7a) is not accessible through SSH. Formally, the safety property to be checked here is $\Box(\forall pkt \in S.rcvq\,.\,\neg pkt.\text{SSH})$. For each incoming *PacketIn* message from switch A, it checks if the enclosed packet is an SSH one and destined to S. If not, it sends a *PacketOut* message instructing A to forward the packet to S. It also sends a *FlowMod* message to A with a rule that allows packets of the same protocol (not SSH) to reach S. In the opposite case (SSH), it checks (a Boolean flag) whether it had previously sent drop rules for SSH packets to the switches. If not, it sets flag to true, sends a *FlowMod* message with a rule that drops SSH packets to A and drops the packet. Note that this inner block does not have an `else` statement.

A fairly common error is to write a statement at the wrong nesting level (§B-CP4). Such a mistake can be built into the above program by nesting the outer `else` branch in the inner `if` block, such that it is executed any time an SSH-packet is encountered but the SSH drop-rule has already been installed (i.e. flag `f` is true). Now, the SSH drop rule, once installed in switch A, disables immediately a potential $nomatch(A, p)$ with $p.\text{SSH} = true$ that would have sent packet p to the controller, but if it has not yet been installed, a second incoming SSH packet would lead to the execution of the `else` statement of the inner branch. This would violate the property defined above, as p will be forwarded to S[9].

MOCS can uncover this bug because of the correct modelling of the controller request queue and the asynchrony between the concurrent executions of control messages sent before a barrier. Otherwise, the second packet that triggers the execution of the wrong branch would not have appeared in the buffer before

[9] Here, we assume that the controller looks up a static forwarding table before sending *PacketOut* messages to switches.

the first one had been dealt with by the controller. Furthermore, if all rules in messages up to a barrier were installed synchronously, the second packet would be dealt with correctly, so no bug could occur.

Inconsistent update bug. OpenFlow's barrier and barrier reply mechanisms allow for updating multiple network switches in a way that enables *consistent packet processing*, i.e., a packet cannot see a partially updated network where only a subset of switches have changed their forwarding policy in response to this packet (or any other event), while others have not done so. MOCS is expressive enough to capture this behaviour and related bugs. In the topology shown in Figure 7a, let us assume that, by default, switch B drops all packets destined to S. Any attempt to reach S through A are examined separately by the controller and, when granted access, a relevant rule is installed at both switches (e.g. allowing all packets from C destined to S for given source and destination ports). Updates must be consistent, therefore the packet cannot be forwarded by A and dropped by B. Both switches must have the new rules in place, before the packet is forwarded. To do so, the controller, (§B-CP5), upon receiving a *PacketIn* message from the client's switch, sends the relevant rule to switch B (*FlowMod*) along with respective barrier (*BarrierReq*) and temporarily stores the packet that triggered this update. Only after receiving *BarrierRes* message from B, the controller will forward the previously stored packet back to A along with the relevant rule. This update is consistent and the packet is guaranteed to reach S. A (rather common) bug would be one where the controller installs the rules to both switches and at the same time forwards the packet to A. In this case, the packet may end up being dropped by B, if it arrives and gets processed before the relevant rule is installed, and therefore the invariant $\Box\big([drop(pkt, sw)]\,.\,\neg(pkt.dest = S)\big)$, where $[drop(pkt, sw)]$ is a quantifier that binds dropped packets (see definition in §B-CP5), would be violated. For this example, it is crucial that MOCS supports barrier response messages.

5 Conclusion

We have shown that an OpenFlow compliant SDN model, with the right optimisations, can be model checked to discover subtle real-world bugs. We proved that MOCS can capture real-world bugs in a more complicated semantics without sacrificing performance.

But this is not the end of the line. One could automatically compute equivalence classes of packets that cover all behaviours (where we still computed manually). To what extent the size of the topology can be restricted to find bugs in a given controller is another interesting research question, as is the analysis of the number and length of interleavings necessary to detect certain bugs. In our examples, all bugs were found in less than a second.

References

1. Al-Fares, M., Radhakrishnan, S., Raghavan, B.: Hedera: Dynamic Flow Scheduling for Data Center Networks. In: NSDI (2010).
2. Al-Shaer, E., Al-Haj, S.: FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In: SafeConfig (2010).
3. Albert, E., Gómez-Zamalloa, M., Rubio, A., et al.: SDN-Actors: Modeling and verification of SDN programs. In: FM (2018).
4. Baier, C., Katoen, J.P.: Principles Of Model Checking, vol. 950 (2008).
5. Ball, T., Bjørner, N., Gember, A., et al.: VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In: PLDI (2014).
6. Behrmann, G., David, A., Larsen, K.G., et al.: Developing UPPAAL over 15 years. Software: Practice and Experience (2011).
7. Braga, R., Mota, E., Passito, A.: Lightweight DDoS flooding attack detection using NOX/OpenFlow. In: LCN (2010).
8. Canini, M., Venzano, D., Perešíni, P., et al.: A NICE Way to Test Openflow Applications. In: NSDI (2012).
9. Cimatti, A., Clarke, E., Giunchiglia, E., et al.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking (2002).
10. Curtis, A.R., Mogul, J.C., Tourrilhes, J., et al.: DevoFlow: scaling flow management for high-performance networks. SIGCOMM (2011).
11. Dobrescu, M., Argyraki, K.: Software dataplane verification. Communications of the ACM (2015).
12. El-Hassany, A., Tsankov, P., Vanbever, L., et al.: Network-wide configuration synthesis. In: CAV (2017).
13. Fayaz, S.K., Sharma, T., Fogel, A., et al.: Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In: OSDI (2016).
14. Fayaz, S.K., Yu, T., Tobioka, Y., et al.: BUZZ: Testing Context-Dependent Policies in Stateful Networks. In: NSDI (2016).
15. Feamster, N., Rexford, J., Shenker, S., et al.: SDX: A software-defined Internet exchange. Open Networking Summit (2013).
16. Feamster, N., Rexford, J., Zegura, E.: The road to SDN. SIGCOMM Computer Communication Review (2014).
17. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. Journal of Computer and System Sciences (1979).
18. Fogel, A., Fung, S., Angeles, L., et al.: A General Approach to Network Configuration Analysis. NSDI (2015).
19. Handigol, N., Seetharaman, S., Flajslik, M., et al.: Plug-n-Serve: Load-balancing web traffic using OpenFlow. SIGCOMM (2009).
20. Havelund, K., Pressburger, T.: Model checking JAVA programs using JAVA PathFinder. STTT (2000).
21. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering (1997).
22. Holzmann, G.J., Peled, D.: An Improvement in Formal Verification. In: FORTE (1994).
23. Horn, A., Kheradmand, A., Prasad, M.R.: Delta-net: Real-time Network Verification Using Atoms. In: NSDI (2017).
24. Hu, H., Ahn, G.J., Han, W., et al.: Towards a Reliable SDN Firewall. In: ONS (2014).
25. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (2002).

26. Jafarian, J.H., Al-Shaer, E., Duan, Q.: OpenFlow random host mutation: Transparent moving target defense using software defined networking. In: HotSDN (2012).
27. Jain, S., Zhu, M., Zolla, J., et al.: B4: Experience with a Globally-Deployed Software Defined WAN. In: SIGCOMM (2013).
28. Jia, Y.: NetSMC : A Symbolic Model Checker for Stateful Network Verification. In: NSDI (2020).
29. Kazemian, P., Chang, M., Zeng, H., et al.: Real Time Network Policy Checking Using Header Space Analysis. In: NSDI (2013).
30. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: Static checking for networks. In: NSDI (2012).
31. Khurshid, A., Zou, X., Zhou, W., et al.: VeriFlow: Verifying Network-wide Invariants in Real Time. In: NSDI (2013).
32. Li, Y., Yin, X., Wang, Z., et al.: A survey on network verification and testing with formal methods: Approaches and challenges. IEEE Surveys & Tutorials (2019).
33. Mai, H., Khurshid, A., Agarwal, R., et al.: Debugging the data plane with anteater. In: SIGCOMM (2011).
34. Majumdar, R., Deep Tetali, S., Wang, Z.: Kuai: A model checker for software-defined networks. In: FMCAD (2014).
35. McClurg, J., Hojjat, H., Černý, P., et al.: Efficient synthesis of network updates. In: PLDI (2015).
36. McKeown, N., Anderson, T., Balakrishnan, H., et al.: OpenFlow: Enabling Innovation in Campus Networks. SIGCOMM Comput. Commun. Rev. (2008).
37. Patel, P., Bansal, D., Yuan, L., et al.: Ananta: Cloud Scale Load Balancing. SIGCOMM (2013).
38. Peled, D.: All from one, one for all: on model checking using representatives. In: CAV (1993).
39. Plotkin, G.D., Bjørner, N., Lopes, N.P., et al.: Scaling network verification using symmetry and surgery. In: POPL (2016).
40. Pnueli, A., Xu, J., Zuck, L.: Liveness with $(0, 1, \infty)$-counter abstraction. In: CAV (2002).
41. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: FOCS (1976).
42. Sethi, D., Narayana, S., Malik, S.: Abstractions for model checking SDN controllers. In: FMCAD (2013).
43. Shenker, S., Casado, M., Koponen, T., et al.: The future of networking, and the past of protocols. In: ONS (2011), `https://tinyurl.com/yxnuxobt`.
44. Son, S., Shin, S., Yegneswaran, V., et al.: Model checking invariant security properties in OpenFlow. In: IEEE (2013).
45. Stoenescu, R., Popovici, M., Negreanu, L., et al.: SymNet: Scalable symbolic execution for modern networks. In: SIGCOMM (2016).
46. Varghese, G.: Vision for Network Design Automation and Network Verification. In: NetPL (Talk) (2018), `https://tinyurl.com/y2cnhvhf`.
47. Yang, H., Lam, S.S.: Real-time verification of network properties using atomic predicates. IEEE/ACM Transactions on Networking (2016).
48. Zeng, H., Kazemian, P., Varghese, G., et al.: A Survey on Network Troubleshooting. Technical Report TR12-HPNG-061012, Stanford University (2012).
49. Zeng, H., Zhang, S., Ye, F., et al.: Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In: NSDI (2014).
50. Zhang, S., Malik, S.: SAT based verification of network data planes. In: Automated Technology for Verification and Analysis. Springer (2013).

A Safeness

Lemma 1 (Safeness). *For an SDN network model* $\mathcal{M}_{(\lambda,\text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$ *and context* $\text{CTX} = (\text{CP}, \lambda, \varphi)$ *with* $\varphi \in LTL_{\setminus\{\bigcirc\}}$,

$$\alpha \in A \text{ is safe} \iff \bigwedge\nolimits_{i=1}^{3} Safe_i(\alpha)$$

where $Safe_i$*, given in Table 1, are per-row.*

Proof. To show safety we need to show two properties: *independence* (action is independent of any other action) and *invisibility* w.r.t. the context, in particular controller program, topology function and formula φ.

Independence: Recall that two actions α and $\beta \neq \alpha$ are independent iff for any state s such that $\alpha \in A(s)$ and $\beta \in A(s)$:

1. $\alpha \in A\big(\beta(s)\big)$ and $\beta \in A\big(\alpha(s)\big)$
2. $\alpha\big(\beta(s)\big) = \beta\big(\alpha(s)\big)$

(1): It can be easily checked that no safe action disables any other action, nor is any safe action disabled by any other action, so the first condition of independence holds.
(2): For any safe action α and any other action β we can assume already that they meet Condition (1). Let us perform a case analysis on α:

- α is either *brepl*, *recv* or *fwd*:
  To show that any interleaving with any action $\beta \neq \alpha$ leads to the same state, we observe that the changes of packet queues by these actions do not interfere with each other. In cases where a packet is removed from a queue by α (e.g. $\alpha = recv(h, pkt)$ removes from $h.rcvq$) but then inserted into the same queue by β (e.g. $\beta = fwd(sw, pkt, ports)$ where $h \in \lambda(sw, ports)_1$), there is no conflict either, as both actions must have been enabled in the original state in the first place. So no conflicts arise for those α.
- α is $ctrl(pkt, cs)$:
  - If β is not a *ctrl* or *bsync* action, then the same argument as above holds.
  - The interesting cases occur when β is in $\{ctrl(\cdot), bsync(\cdot)\}$. From $Safe_2(\alpha)$ we know that CP is not order-sensitive, which implies that α and β are independent. Order-insensitivity is a relatively strong condition but it ensures correctness of the lemma and thus partial order reduction.[10] Thus any interleaving of α and β leads to the same state.
- α is $bsync(sw, xid, cs)$:
  The same line of argument applies as for $ctrl(pkt, cs)$, simply exchanging the roles of α and β.

[10] Generalisations by a more clever analysis of the controller program are a future research topic.

Invisibility : We show this for all safe actions separately:

- $\alpha = ctrl(pk, cs)$. The only variables α can change are the $controller.rq$, $sw'.fq$, $sw'.cq$ (for some switches sw'), and the control state cs. The first three can not appear in φ due to the definition of the specification language. In case the control state changes, α is invisible to φ because $Safe_3(\alpha)$ in Table 1.
- $\alpha = bsync(sw, xid, cs)$. This α only affects brq, $sw'.fq$, $sw'.cq$ (for some switches sw'), and the control state cs. We know by definition of Specification Language (§2.4) that it cannot refer to brq or any $sw'.fq$, $sw'.cq$. In case the control state changes, α is invisible to φ because $Safe_3(\alpha)$ in Table 1.
- $\alpha = fwd(sw, pk, ports)$. Assumption $Safe_3(\alpha)$ in Table 1 guarantees that the only variables α can change, i.e. $D.pq$ or $D.rcvq$ for any D in $\lambda(sw, p)_1 \mid p \in ports$ and $sw.pq$, actually remain unchanged. Thus it follows by definition that α is invisible to φ.
- $\alpha = brepl(sw, xid)$. Since, by definition of Specification Language (§2.4), the atomic propositions refer neither to any cq nor brq, it follows from the effect of α that only affects $sw.cq$ and brq that any $brepl(\cdot)$ is always invisible.
- $\alpha = recv(h, pk)$. Assumption $Safe_3(\alpha)$ in Table 1 guarantees that φ does not refer to $h.rcvq$, which is the only variable affected by α, and therefore $recv(h, pk)$ is invisible to φ.

□

B Controller Programs

```
1  handler pktIn(sw, pkt):
2  |  if not pkt.SSH then                                          // Otherwise, pkt is dropped silently
3  |  |  send_message(PacketOut(pkt, 2), sw)
4  |  end
5  |  rule1 ← {{prio ← 10}, {SSH  ← 1  }, {in_port ← *}, {fwd_port ← drop}}
6  |  rule2 ← {{prio ←  1}, {SSH  ← *  }, {in_port ← 1}, {fwd_port ← 2   }}      // asterisk (*) matches any value
7  |  rule3 ← {{prio ←  1}, {SSH  ← *  }, {in_port ← 2}, {fwd_port ← 1   }}
8  |  forall s ∈ Switches do                                       // Switches is the set of all switches
9  |  |  send_message(FlowMod(add(rule2)), s)
10 |  |  send_message(FlowMod(add(rule1)), s)
11 |  |  send_message(BarrierReq(b_id),    s)                        // b_id is a barrier identifier
12 |  |  send_message(FlowMod(add(rule3)), s)
13 |  end
```

Controller Program CP1: A stateless firewall filter with control messages reordering bug. In a bug-free program (the one we used to verify in §4), `rule1` should be sent first and followed by a barrier. Property: *"neither host should be accessed over* SSH*"*. Formally, $\square(\forall h \in Hosts\, \forall pkt \in h.rcvq\,.\, \neg pkt.\text{SSH})$.

```
handler pktIn(sw, pkt):
    if allowed_conn[pkt.src][pkt.src_TCP_port][pkt.dest][pkt.dest_TCP_port] then          /* allowed_conn is a fixed
                                                                                            * whitelist of TCP
                                                                                            * socket connections
                                                                                            * (host, TCP_port) ↦
                                                                                            * (host, TCP_port)
                                                                                            */
        send_message(PacketOut(pkt, 2), sw)
        rule1.src           ← pkt.src
        rule1.src_TCP_port  ← pkt.src_TCP_port
        rule1.dest          ← pkt.dest
        rule1.dest_TCP_port ← pkt.dest_TCP_port
        rule1.fwd_port      ← 2
        rule1.prio          ← 2
        rule2.src           ← pkt.dest
        rule2.src_TCP_port  ← pkt.dest_TCP_port
        rule2.dest          ← pkt.src
        rule2.dest_TCP_port ← pkt.src_TCP_port
        rule2.fwd_port      ← 1
        rule2.prio          ← 2
        forall s ∈ Switches do // access rules are uniform across all switches, any of which acting as firewall replica
            send_message(FlowMod(add(rule1)), s)
            send_message(FlowMod(add(rule2)), s)
            send_message(BarrierReq(b_id),    s)            // b_id is uniquely associated with an allowed connection
        end
    else
        send_message(PacketOut(pkt, drop), sw)
        drop_rule.src           ← pkt.src
        drop_rule.src_TCP_port  ← pkt.src_TCP_port
        drop_rule.dest          ← pkt.dest
        drop_rule.dest_TCP_port ← pkt.dest_TCP_port
        drop_rule.fwd_port      ← drop
        drop_rule.prio          ← 1
        forall s ∈ Switches do
            send_message(FlowMod(add(drop_rule)), s)          // access restrictions are uniform across all replicas
        end
    end

handler barrierIn(sw, xid):
    controller_view[b_id][sw] ← true    /* controller_view associates installed rules (through the respective b_id)
                                           * for respective allowed connections with switches
                                           */
```

Controller Program CP2: Stateful inspection firewall (Figure 7b). The property we verify is: *"a packet is never dropped by a rule in a switch if the controller is aware of a matching rule being already installed in this switch"*. Formally: $\Box\Big([drop_m(pkt, sw)]\neg controller\_view[pkt.src][pkt.src\_TCP\_port][pkt.dest][pkt.dest\_TCP\_port][sw]\Big)$ where $[drop_m(pkt, sw)]P$ is short for $[match(sw, pkt, r)]\big((r.fwd\_ports = \texttt{drop}) \Rightarrow P\big)$.

```
handler pktIn(sw, pkt):
    if not MAC_table[sw][pkt.src] then                          // MAC_table associates sender with a switch port
        MAC_table[sw][pkt.src] ← pkt.in_port
    end
    if MAC_table[sw][pkt.dest] then
        send_message(PacketOut(pkt, MAC_table[sw][pkt.dest]), sw)
        rule.src      ← pkt.src
        rule.dest     ← pkt.dest
        rule.in_port  ← pkt.in_port
        rule.fwd_port ← MAC_table[sw][pkt.dest]
        rule.prio     ← 1
        send_message(FlowMod(add(rule)), sw)
    else
        send_message(PacketOut(pkt, flood\{pkt.in_port}), sw) // pkt will be flooded to all ports except incoming one
    end
```

Controller Program CP3: MAC learning application[†] for verifying absence of loops. In order to keep track of the network devices the packet passes through (i.e. the packet path history), the packet type is augmented with a history bit-field *reached*, where each bit represents a visited/unvisited switch. As packets are being flooded, their history bit-field is re-written. The loop freedom property asserts that *"a packet should not come back to the same switch"*. Formally, $\Box\big(\forall sw \in Switches\ \forall pkt \in sw.pq\,.\,\neg pkt.reached[sw]\big)$.

[†] https://github.com/noxrepo/pox/blob/412a6adb38cb646748c8cfb657549787ab6d2e88/pox/forwarding/l2_learning.py

```
1  handler pktIn(sw, pkt):
2  |  if pkt.SSH and pkt.dest == S then
3  |  |  if not f then                                          // f is initialised as false. pkt is dropped silently
4  |  |  |  f                  ← true
5  |  |  |  drop_rule.prio     ← 1
6  |  |  |  drop_rule.SSH      ← pkt.SSH
7  |  |  |  drop_rule.dest     ← pkt.dest
8  |  |  |  drop_rule.fwd_port ← drop
9  |  |  |  forall s ∈ Switches do
10 |  |  |  |  send_message(FlowMod(add(drop_rule)), s)
11 |  |  |  |  send_message(BarrierReq(b_id),        s)          // b_id is a barrier identifier
12 |  |  |  end
13 |  |  else
14 |  |  |  send_message(PacketOut(pkt, 2), sw)
15 |  |  |  rule.prio      ← 2
16 |  |  |  rule.SSH       ← pkt.SSH
17 |  |  |  rule.dest      ← pkt.dest
18 |  |  |  rule.fwd_port ← 2
19 |  |  |  forall s ∈ Switches do
20 |  |  |  |  send_message(FlowMod(add(rule)), s)
21 |  |  |  end
22 |  |  end
23 |  else
24 |  |  ...
25 |  end
```

Controller Program CP4: Wrong nesting level bug: Executing the `else`-branch - shaded red - would violate the policy that *"server S (Figure 7a) should not be accessed over* SSH*"*, $\square(\forall pkt \in S.rcvq\,.\,\neg pkt.\text{SSH})$.

```
1  handler pktIn(sw, pkt):            // Assumption: a drop-all rule with priority 0 is installed in switch B (Fig.7a)
2  |  if pkt.dest == S and BarrierRes(b_id) not received then        /* b_id is uniquely associated with rule_S which
                                                                       * overrides the drop-all entry at B, and
                                                                       * allows packets to be forwarded to S through
                                                                       * port 2
                                                                       */
3  |  |  if not packets_held[sw][pkt] then   /* packets_held is temporarily storing packets sent by B until consistent
                                               * update is complete
                                               */
4  |  |  |  packets_held[sw][pkt] ← true
5  |  |  |  rule_S ← {{dest ← S}, {fwd_port ← 2}, {prio ← 2}}
6  |  |  |  send_message(FlowMod(add(rule_S)), B)
7  |  |  |  send_message(BarrierReq(b_id),     B)
8  |  |  end
9  |  else
10 |  |  send_message(PacketOut(pkt, 2), sw)
11 |  end
12
13 handler barrierIn(sw, xid):
14 |  if xid == b_id then
15 |  |  rule_S ← {{dest ← S}, {fwd_port ← 2}, {prio ← 2}}
16 |  |  forall s ∈ Switches\{B} do                                  // all switches except B
17 |  |  |  send_message(FlowMod(add(rule_S)), s)
18 |  |  end
19 |  |  while packets_held[swi][p] for some (p, swi) and p.dest == S do
20 |  |  |  packets_held[swi][p] ← false                          // swi is the switch packet p was sent from
21 |  |  |  send_message(PacketOut(p, 2), swi)
22 |  |  end
23 |  end
```

Controller Program CP5: Consistent updates. We verify the property that *"a packet destined to server S is never dropped at any switch"*. Formally: $\square\big([drop_{mf}(pkt, sw)]\neg(pkt.dest = S)\big)$, where $[drop_{mf}(pkt, sw)]P$ is short for $[match(sw, pkt, r)]\big((r.fwd\_ports = \texttt{drop}) \Rightarrow P\big) \wedge [fwd(sw, pkt, fwd\_ports)]\big((fwd\_ports = \texttt{drop}) \Rightarrow P\big)$.

Chapter 4

Model Checking Software-Defined Networks with Flow Entries that Time Out

This chapter is an extended version of the author's paper "Model Checking Software-Defined Networks with Flow Entries that Time Out" in Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design – FMCAD 2020. The chapter builds upon the framework model presented in Chapter 3. To comply with OpenFlow protocol, it extends this model by allowing flow entries installed in flow tables of switches to expire. This is an essential extension for modelling TCP sockets that can timeout, i.e., modelling timeouts in which a socket expects to receive an acknowledgement for sent data before it decides that the connection has failed. To counterbalance the effect of increasing expressivity, the model is further optimised by identifying new safe actions which do not have to be interleaved with other ones, pruning thus redundant state-explorations (as in Chapter 3). We evaluate the performance of the proposed model extensions in a dual-mixed setting using a load balancer and a firewall in network topologies of varying size.

Model Checking Software-Defined Networks with Flow Entries that Time Out

Vasileios Klimis, George Parisis and Bernhard Reus
University of Sussex, UK
{v.klimis, g.parisis, bernhard}@sussex.ac.uk

***Abstract*—Software-defined networking (SDN) enables advanced operation and management of network deployments through (virtually) centralised, programmable controllers, which deploy network functionality by installing rules in the flow tables of network switches. Although this is a powerful abstraction, buggy controller functionality could lead to severe service disruption and security loopholes, motivating the need for (semi-)automated tools to find, or even verify absence of, bugs. Model checking SDNs has been proposed in the literature, but none of the existing approaches can support dynamic network deployments, where flow entries expire due to timeouts. This is necessary for automatically refreshing (and eliminating stale) state in the network (termed as *soft-state* in the network protocol design nomenclature), which is important for scaling up applications or recovering from failures. In this paper, we extend our model (MoCS) to deal with timeouts of flow table entries, thus supporting soft state in the network. Optimisations are proposed that are tailored to this extension. We evaluate the performance of the proposed model in UPPAAL using a load balancer and firewall in network topologies of varying size.**

I. INTRODUCTION

Software-defined networking (SDN) [1] revolutionised network operation and management along with future protocol design; a virtually centralised and programmable controller 'programs' network switches through interactions (standardised in OpenFlow [2]) that alter switches' flow tables. In turn, switches push packets to the controller when they do not store state relevant to forwarding these packets. Such a paradigm departure from traditional networks enables the rapid development of advanced and diverse network functionality; e.g., in designing next-generation inter-data centre traffic engineering [3], load balancing [4], firewalls [5] and Internet exchange points (IXPs) [6]. Although this is a powerful abstraction, buggy controller functionality could lead to severe service disruption and security loopholes. This has led to a significant amount of research on SDN verification and/or bug finding, including static network analysis [7], [8], [9], dynamic real-time bug finding [10], [11], [12], [13], and formal verification approaches, including symbolic execution [14], [15], [16] and model checking [17], [10], [16], [18] methods. A comprehensive review of existing approaches along with their shortcomings can be found in [19].

Model checking is a renowned automated technique for hardware and software verification and existing model checking approaches for SDNs have shown promising results with respect to scalability and model expressivity, in terms of supporting realistic network deployments and the OpenFlow standard. However, a key limitation of all existing approaches is that they cannot model forwarding state (added in network switches' flow tables by the controller) that expires and gets deleted. Without this, one cannot model nor verify the correctness of SDNs with soft-state which is prominent in the design of protocols and systems that are resilient to failures and scalable; e.g., as in [20], where flow scheduling is on a per-flow basis, and numerous network protocols where in-network state is not explicitly removed but expires, so that overhead is minimised [21].

In this paper, we extend our model (MoCS) [17] to support soft-state, complying with the OpenFlow specification, by allowing flow entries to time out and be deleted. We propose relevant optimisations (as in [17]) in order to improve verification performance and scalability. We evaluate the performance of the proposed model extensions in UPPAAL using a load balancer and firewall in network topologies of varying size.

II. MOCS SDN MODEL

The MoCS model [17] is formally defined by means of an action-deterministic transition system. We parameterise the model by the underlying network topology, λ, and the controller program, CP, in use. The model is a 6-tuple $\mathcal{M}_{(\lambda,\text{CP})} = (S, s_0, A, \hookrightarrow, AP, L)$, where S is the set of all states the SDN may enter, s_0 the initial state, A the set of actions which encode the events the network may engage in, $\hookrightarrow \subseteq S \times A \times S$ the transition relation describing which execution steps the system undergoes as it perform actions, *AP* a set of atomic propositions describing relevant state properties, and $L : S \to 2^{AP}$ is a labelling function, which relates to any state $s \in S$ a set $L(s) \in 2^{AP}$ of those atomic propositions that are true for s. Such an SDN model is composed of several smaller systems, which model network components (hosts[1], switches and the controller) that communicate via queues and, combined, give rise to the definition of $\hookrightarrow$. A detailed description of MoCS' components and transitions can be found in [17]. Due to lack of space, in this paper, we only discuss aspects of the model that are required to understand and verify the soundness of the proposed model extensions, and examples used in the evaluation section. Figure 1 illustrates a high-level view of OpenFlow interactions, modelled actions and queues, including the proposed extensions discussed in Section III.

[1]A host can act as a client and/or server.

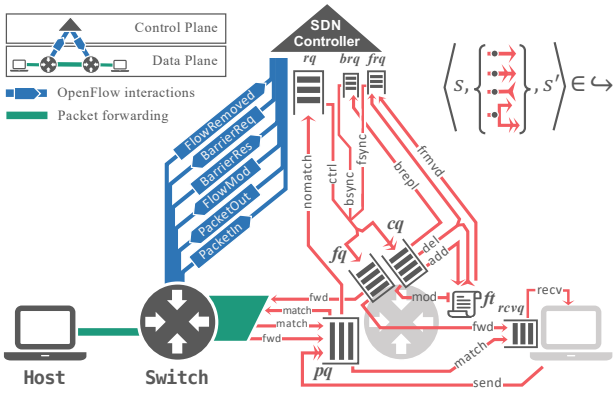


Fig. 1: A high-level view of OpenFlow interactions (left half) and modelled actions (right half). A red solid-line arrow depicts an action which, when fired, (1) dequeues an item from the queue the arrow begins at, and (2) (possibly) adds an item in the queue the arrowhead points to (or multiple items if the arrow is double-headed). Deleting an item from the target queue is denoted by a reverse arrowhead; modifying in, by a hammerhead. A forked arrow denotes (possibly) multiple targeted queues.

States and queues: A *state* is a triple (π, δ, γ), where π is a family of hosts, each consisting of a receive queue (*rcvq*); δ is a family of switches, consisting of a switch packet queue (*pq*), switch forward queue (*fq*), switch control queue (*cq*), switch flow table (*ft*); γ consists of the local controller program state $cs \in CS$, and a family of controller queues: request queue (*rq*), barrier-reply queue (*brq*) and flow-removed queue (*frq*). So π and δ describe the *data-plane*, and γ the *control plane*. The network components communicate via the shared queues. Each transition models a certain network event that will involve some of the queues, and maybe some other network state. Concurrency is modelled through interleavings of those events.
Transitions: Each transition is labelled with an action $\alpha \in A$ that indicates the nature of the network event. We write $s \xhookrightarrow{\alpha} s'$ and $(s, \alpha, s') \in \hookrightarrow$ interchangeably to denote that the network moved from state s to s' by executing transition α. The parts of the network involved in each individual α, i.e. *packets, rules, barriers, switches, hosts, ports* and *controller states*, are included in the transition label as parameters; e.g., $match(sw, pkt, r) \in A$ denotes the action that switch sw matches packet pkt by rule r and, as a result, forwards it accordingly, leading to a new state after transition.
Atomic propositions: The propositions in *AP* are statements on (1) controller program states, denoted by $Q(q)$ which expresses that the controller program is in state $q \in CS$, allowing one to reason about the controller's internal data structures, and (2) packet header fields – those packets may be in any switch buffer *pq* or host buffer *rcvq* (but no other buffers). For instance, $\exists pkt \in sw.pq\,.\,P(pkt)$ is a legitimate atomic proposition that states that there is a packet in sw's packet queue that satisfies packet *pkt* property P.
Topology: λ describes the network topology as a bijective map which associates one network interface (a pair of networking device and physical port) to another.
Specification Logic: The properties of the SDNs to be checked in this paper are safety properties, expressed in linear-time temporal logic without 'next-step' operator, $\text{LTL}_{\setminus\{\bigcirc\}}$. We have enriched the logic by modal operators of *dynamic logic* [22], allowing formula construct of the form $[\alpha(\vec{x})]P$ stating that whenever an event $\alpha(\vec{x})$ happened, P must hold. Note that P may contain variables from x. This extension is syntax sugar in the sense that the formulae may be expressed by additional state; e.g., $\big[match(sw, pkt, r)\big](r.fwdPort = \texttt{drop})$ states that if *match* happened, it was via a rule that dropped the packet. This permits specification formulae to be interpreted not only over states, but also over actions that have happened. The model checking problem then, for an SDN model $\mathcal{M}_{(\lambda,\text{CP})}$ with a given topology λ, a control program CP and a formula φ of the specification logic as described above, boils down to checking whether all runs of $\mathcal{M}_{(\lambda,\text{CP})}$ satisfy φ, short $\mathcal{M}_{(\lambda,\text{CP})} \models \Box\varphi$.
SDN Operation: End-hosts send and receive packets (*send* and *recv* actions in Figure 1) and switches process incoming packets by matching them (or failing to) with a flow table entry (rule). In the former case (*match* action), the packet is forwarded as prescribed by the rule. In the opposite case (*nomatch* action), the packet is sent to the controller (*PacketIn* message on the left side of Figure 1). The controller's packet handler is executed in response to incoming *PacketIn* messages; as a result of its execution, its local state may change, a number of packets (*PacketOut* message) and rule updates (*FlowMod* message), interleaved with barriers (*BarrierReq* message), may be sent to network switches. Network switches react to incoming controller messages; they forward packets sent by the controller as specified in the respective *PacketOut* message (*fwd* action), update their own forwarding tables (*add/del* actions), respecting set barriers and notifying the controller (*BarrierRes* message) when said barriers are executed (*brepl* action). Finally, upon receiving a *BarrierRes* message, the controller executes the respective handler (*bsync* action), which can result in the same effects as the *PacketIn* message handler.
Abstractions: To obtain finitely representable states, all queues in the model must be finitely representable. For packet queues we use multisets, subject to $(0, \infty)$ abstraction [23]; a packet either does not appear in the queue or appears an unbounded number of times. The other queues are simply modelled as finite sets. Modelling queues as sets means that entries are not processed in the order of arrival. This is intentional for packet queues but for controller queues this may limit behaviour unless the controller program is order-insensitive. We focus on those controller programs in this paper.

III. Modelling Flow Entry Timeouts

In order to model soft-state in the network, we enrich our model with two new actions that model flow entry timeouts and subsequent handling of these timeouts by the controller

program. Note that in our model, timeouts are not triggered by any kind of clock; instead, they are modelled through the interleaving of actions in the underlying transition system that ensure that flow removal (and subsequent handling by the controller program) will appear as it would for any possible value of a timeout in a real system.

The new actions are defined as follows: $frmvd(sw, r)$ models the timeout event, as an action in the transition system that removes the flow entry (rule) r from switch sw and notifies the controller by placing a *FlowRemoved* message (see Figure 1) in the respective queue (*frq*). The $fsync(sw, r, cs)$ action models the call to the *FlowRemoved* message handler. As a result of the handler execution, the controller's local state (*cs*) may change, a number of packets (*PacketOut* messages) and rule updates (*FlowMod* messages), interleaved with barriers (*BarrierReq* message), may be sent to network switches. In order to model timeouts, rules are augmented with a *timeout* bit which, when true, signals that the installed rule can be removed at any time, i.e., the *frmvd*-action can be interleaved, in any order, with any other action that is enabled at any state later than the installation of this rule.

To support our examples, we add to the set of *FlowMod* messages a *modify flow entry* instruction. In [17] we only used $add(sw, r)$ and $del(sw, r)$ messages, for installing and deleting rule r at switch sw, respectively. We now add $mod(sw, f, a)$ to these messages. This instructs switch sw that if a rule is found in $sw.ft$ that matches field *f*, its forwarding actions are modified by *a*. If no such rule exists, $mod(\cdot)$ does not do anything.

Optimisation: To tackle the state-space explosion, we exploit the fact that some traces are observationally (w.r.t. the property to be proved) equivalent, so that only one of those needs to be checked. This technique, referred to as *partial-order reduction* (POR) [24], reduces the number of interleavings (traces) one has to check. To prove equivalence of traces, one needs actions to be permutable and invisible to the property at hand. This is the motivation for the following definition:

Definition 1 (SAFE ACTIONS) Given a context $\text{CTX} = (\text{CP}, \lambda, \varphi)$, and SDN model $\mathcal{M}_{(\lambda,\text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$, an action $\alpha(\cdot) \in A(s)$ is called *safe* if it is (1) *independent* of any other action β in A, i.e. executing α after β leads to the same state as running β after α, and (2) *unobservable* for φ (also called *φ-invariant*), i.e., $s \models \varphi$ iff $\alpha(s) \models \varphi$ for all $s \in S$ with $\alpha \in A(s)$.

The following property of controller programs is needed to show safety:

Definition 2 (ORDER-SENSITIVE CONTROLLER PROGRAM) A controller program CP is order-sensitive if there exists a state $s \in S$ and two actions α, β in $\{ctrl(\cdot), bsync(\cdot), fsync(\cdot)\}$ such that $\alpha, \beta \in A(s)$ and $s \overset{\alpha}{\hookrightarrow} s_1 \overset{\beta}{\hookrightarrow} s_2$ and $s \overset{\beta}{\hookrightarrow} s_3 \overset{\alpha}{\hookrightarrow} s_4$ with $s_2 \neq s_4$.

In [17] we already showed that certain actions are safe and can be used for PORs. We now show that the new $fsync(\cdot)$ action is safe on certain conditions.

Lemma 1 (SAFENESS PREDICATES FOR *fsync*) For transition system $\mathcal{M}_{(\lambda,\text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$ and a formula $\varphi \in \text{LTL}_{\setminus\{\bigcirc\}}$, $\alpha = fsync(sw, r, cs)$ is safe iff the following two conditions are satisfied:

Independence CP is not order-sensitive
Invisibility `if` $Q(q)$ in AP occurs in φ, `then` α is φ-invariant

Proof. See Appendix A. □

Given a context $\text{CTX} = (\text{CP}, \lambda, \varphi)$ and an SDN network model $\mathcal{M}_{(\lambda,\text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$, for each state $s \in S$ define $ample(s)$ as follows: if $\{\alpha \in A(s) \mid \alpha \text{ safe }\} \neq \varnothing$, then $ample(s) = \{\alpha \in A(s) \mid \alpha \text{ safe }\}$; otherwise $ample(s) = A(s)$. Next, we define $\mathcal{M}^{fr}_{(\lambda,\text{CP})} = (S^{fr}, A, \hookrightarrow_{fr}, s_0, AP, L^{fr})$, where $S^{fr} \subseteq S$ the set of states reachable from the initial state s_0 under $\hookrightarrow_{fr}$, $L^{fr}(s) = L(s)$ for all $s \in S^{fr}$ and $\hookrightarrow_{fr} \subseteq S^{fr} \times A \times S^{fr}$ is defined inductively by the rule:

$$\frac{s \overset{\alpha}{\hookrightarrow} s'}{s \overset{\alpha}{\hookrightarrow}_{fr} s'} \quad \text{if} \quad \alpha \in ample(s)$$

Now we can proceed to extend the POR Theorem of [17]:

Theorem 1 (FLOW-REMOVED EQUIVALENCE) Given a property $\varphi \in \text{LTL}_{\setminus\{\bigcirc\}}$, it holds that $\mathcal{M}^{fr}_{(\lambda,\text{CP})}$ satisfies φ iff $\mathcal{M}_{(\lambda,\text{CP})}$ satisfies φ.

The proof is a consequence of Lemma 1 applied to the proof of Theorem 2 in [17]. See Appendix A for a detailed proof.

IV. EXPERIMENTAL EVALUATION

In this section we experimentally evaluate the proposed extensions in terms of verification performance and scalability. We use a realistic controller program that enables a network switch to act both as a load balancer and stateful firewall (see §V-CP1). The load balancer keeps track of the active sessions between clients and servers in the cluster (see Figure 2), while, at the same time, only allowing specific clients to access the cluster. Soft state is employed here so that flow entries for completed sessions (that were previously admitted by the firewall) time out and are deleted by the switch without having to explicitly monitor the sessions and introduce unnecessary signalling (and overhead). In the underlying SDN model, the *frmvd* action is fired, which, in turn, deletes the flow entry from the switch's table and notifies the controller of that. This enables the *fsync* action that calls the flow removal handler.

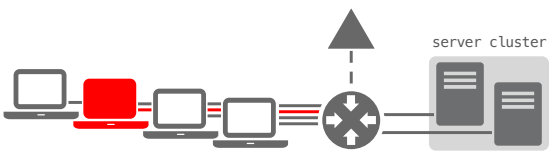


Fig. 2: Four clients and two servers connecting to an OF-switch. ■ is not white-listed.

A session is initiated by a client which sends a packet (*pkt* in §V-CP1) to a known cluster address; servers are

not directly visible to the client. Sessions are bi-directional therefore the controller must install respective rules to the switch to allow traffic to and from the cluster. The property that is checked here is that (1) the traffic (i.e. number of sessions, assuming they all produce similar traffic patterns), and resulting load, is uniformly distributed to all available servers, and (2) that traffic from non-whitelisted clients is blocked. More concretely, "*a packet from a 'dodgy' address should never reach the servers, and the difference between the number of assigned sessions at each server should never be greater than 1*", formally,

$$\Box\ \big(\forall s_i, s_j \in \mathit{Servers}\ \forall \mathit{pkt} \in s_i.\mathit{rcvq}\ .\ \neg \mathit{pkt}.\mathit{src} = \mathit{dodgy}\ \wedge\ \big|\mathit{sLoad}[s_i] - \mathit{sLoad}[s_j]\big| <\ \ 2\big) \qquad (\varphi)$$

where *sLoad* stores the active session count for each server.

In the first (buggy) version of the controller's packet handler (shaded grey in §V-CP1) and flow removal handler §V-CP2, the controller program assigns new sessions to servers in a round-robin fashion and keeps track of the active sessions (array *deplSessions* in the provided pseudocode). When a session expires, the respective flow table entry is expected to expire and be deleted by the switch without any signalling between the controller, clients or servers[2]. As stated above, this controller program does not satisfy safety property φ because the controller does nothing to rebalance the load when a session expires. Our model implementation[3] discovered the bug in the topology shown in Figure 2 with 3 sessions in 11ms exploring 202 states.

In the second (still buggy) version of the controller, session scheduling is more sophisticated (shaded blue in §V-CP1); a session is assigned to the server with the least number of active sessions. Although the updated load balancing algorithm does keep track of the active sessions per server, this controller is still buggy because no rebalancing takes place when sessions expire. In a topology of 4 clients and 2 servers, we were able to discover the bug in 52ms after exploring 714 states.

We fix the bug by allowing the controller program to rebalance the active sessions, when (1) a session expires and (2) the load is about to get out of balance, by moving one session from the most-loaded to the least-loaded server (§V-CP3). In the same topology as above, we verified the property in 625ms after exploring 15068 states.[4]

Next, we evaluate the performance of the proposed model and extensions for verifying the correctness of the property in a given SDN. We do that by verifying φ with the correct controller program, discussed above, and scaling up the topology in terms of clients, servers and active sessions. Results are listed in Table I and state exploration is illustrated in Figure 4.

Table I lists performance of the model checker for verifying the correct controller program with PORs disabled on the left and with PORs enabled on the right, respectively. For each chosen topology we list the number of states explored, CPU time used, and memory used. The topology is shaped as in Figure 2, and parametrised by the number of clients (ranging from 3 to 5) and servers (ranging from 2 to 5), as indicated in Table I. The number of required packets and rules, respectively, is shown in grey. These numbers are always uniquely determined by the choice of topology. Where there are no entries in the table (indicated by a dash) the verification did not terminate within 24 hours.

The results clearly show that the verification scales well with the number of servers but not with the number of clients. The reason for the latter is that for each additional client an additional packet is sent, which, according to programs §V-CP1 and CP3, leads to 7 additional actions without timeouts and to 12 with timeouts. The causal ordering of these actions is shown in Fig. 3. The sub-branch in red shows the actions that appear due to a timeout of the added rule. Thus, the number of states is exponential in the number of clients: every new action in Fig. 3 leads to a new change of state, thus doubling the possible number of states. This exponential blow-up happens whether we have timeouts or not. With timeouts, however, we have worse exponential complexity as there are more new states generated.

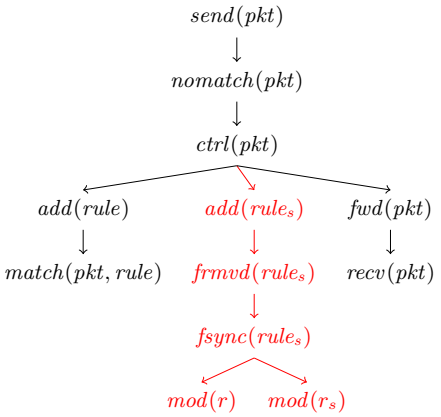


Fig. 3: The causal enabling relation between actions for an additional packet *pkt*; only the relevant arguments are shown using the same nomenclature as in the pseudocode.

The results also demonstrate that, for network setups with three clients, the POR optimisation reduces the state space – and thus the verification time – by about half. For more clients the reduction is far more significant, given that the verification of the unoptimised model did not terminate within 24 hours. This is not surprising as the number of possible interleavings is massively increased by the non-deterministic timeout events.

V. Controller Programs

CP1 implements the *PacketIn* message handler that processes packets sent by switches when the *nomatch* action is fired. The two different versions of functionality discussed in the paper are defined by the *leastConnectionsScheduling*

[2]It is worth stressing that modelling such functionality is not supported by existing model checking approaches, such as [17] and [18], where flow table entries can only be explicitly deleted by the controller.

[3]Uppaal [25] is the back-end verification engine for MoCS and all experiments were run on an 18-Core iMac pro, 2.3GHz Intel Xeon W with 128GB DDR4 memory.

[4]Note that the *fsync*-optimisation was not enabled in the examples above.

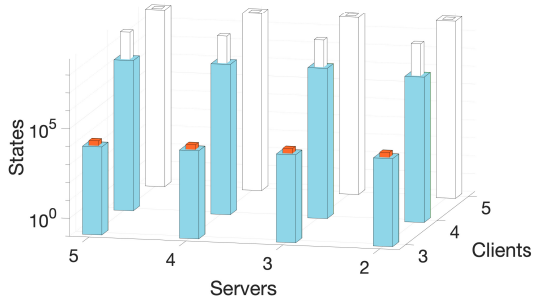


Fig. 4: Explored States (logarithmic scale). Wide bars represent the optimised model and narrow ones (inside) the unoptimised model. Uncoloured bars represent non-termination.

TABLE I: Performance by number of clients and servers

Clients	Servers	Packets	Rules	without POR States	without POR CPU user time	without POR Resident memory [KiB]	with POR States	with POR CPU user time	with POR Resident memory [KiB]
3	2	3	13	15,068	553ms	9,516	8,264	317ms	9,016
3	3	3	19	15,068	700ms	10,688	8,264	322ms	8,792
3	4	3	25	15,068	841ms	11,936	8,264	483ms	10,488
3	5	3	31	15,068	987ms	15,280	8,264	563ms	12,844
4	2	4	17	–	–	–	13,244,474	13.2m	2,508,528
4	3	4	25	–	–	–	24,623,435	30.77m	5,432,004
4	4	4	33	–	–	–	24,623,435	37.23m	13,129,916
4	5	4	41	–	–	–	24,623,435	42.64m	15,443,136
5	2	5	21	–	–	–	–	–	–

constant. When *leastConnectionsScheduling* is false, server selection is done in a round-robin fashion, whereas, in the opposite case, the controller assigns the new session to the server with the least number of active sessions.

CP2 implements the naive (and buggy) *FlowRemoved* message handler. When soft state expires in the network, the handler merely updates its local state to reflect the update in the load.

CP3 implements a more sophisticated (and correct) *FlowRemoved* message handler. When soft state expires in the network, the handler updates its local state to reflect the update in the load and re-assigns active sessions from the most to the least loaded server, by updating the flow table of the switch accordingly.

VI. Conclusion and Future Work

We have proposed model checking of SDN networks with flow entries (rules) that time out. Timeouts pose problems due to the great number of resulting interleavings to be explored. Our approach is the first one to deal with timeouts, exploiting partial-order reductions, and performing reasonably well for small networks. We demonstrated that bug finding works well for SDN networks in the presence of flow entry timeouts. Future work includes exploring flow removals with timeouts that are constrained by integer to *enforce certain orderings* of timeout messages as well as improvements in performance, for instance, by using bounded model checking tools for concurrent programs.

Controller Program CP 1: *PacketIn* Message Handler

```
1:  handler pktIn(pkt, sw)
2:      if pkt.srcIP ≠ dodgy_client then
3:          if ¬deplSessions[pkt.srcIP] then
4:              if ¬leastConnectionsScheduling then
                    // Round-Robin rotation
5:                  server ← server mod 2 + 1
6:              else
                    // Least-Connections scheduling
7:                  server ← min(sLoad[])
8:              end if
                // Initialisation of flow to server
9:              rule.srcIP          ← pkt.srcIP
10:             rule.in_port        ← pkt.in_port
11:             rule.fwdPort        ← server
                // Initialisation of symmetric rule_s
12:             rule_s.srcIP        ← server
13:             rule_s.destIP       ← pkt.srcIP
14:             rule_s.fwdPort      ← pkt.in_port
15:             rule_s.timeout      ← true
                // Initialisation of drop rule rule_d
16:             rule_d.srcIP        ← dodgy_client
17:             rule_d.fwdPort      ← drop
                // Deployment of rules
18:             send_message(FlowMod(add(rule)), sw)
19:             send_message(FlowMod(add(rule_s)), sw)
20:             send_message(FlowMod(add(rule_d)), sw)
                // Update firewall state table
21:             sLoad[server]++
22:             deplSessions[pkt.srcIP] ← true
23:         end if
            // PacketOut: sending pkt out through sw
24:         send_message{PacketOut(pkt, server), sw)
25:     end if
26: end handler
```

Controller Program CP 2: Naive *FlowRemoved* message handler

```
1: handler flowRmvd(rule_s, sw)
2:     sLoad[rule_s.srcIP]--
3:     deplSessions[rule_s.destIP] ← false
4: end handler
```

Controller Program CP 3: Correct *FlowRemoved* message handler

```
1:  handler flowRmvd(rule_s, sw)
2:      sLoad[rule_s.srcIP]--
3:      deplSessions[rule_s.destIP] ← false
4:      if max(sLoad[]) − min(sLoad[]) > 1 then
5:          r ← the rule in sw.ft with fwdPort = max(sLoad[])
6:          r_s ← symmetric rule of r
7:          cm ← mod(r, fwdPort ← min(sLoad[]))
8:          cm_s ← mod(r_s, srcIP ← min(sLoad[]))
9:          send_message(FlowMod(cm, sw))
10:         send_message(FlowMod(cm_s, sw))
11:         sLoad[max(sLoad[])]--
12:         sLoad[min(sLoad[])]++
13:     end if
14: end handler
```

REFERENCES

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN," *SIGCOMM Computer Communication Review*, 2014.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, 2008.

[3] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: scaling flow management for high-performance networks," *SIGCOMM*, 2011.

[4] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing web traffic using OpenFlow," *SIGCOMM*, 2009.

[5] H. Hu, G.-J. Ahn, W. Han, and Z. Zhao, "Towards a Reliable SDN Firewall," in *ONS*, 2014.

[6] N. Feamster, J. Rexford, S. Shenker, R. Clark, R. Hutchins, D. Levin, and J. Bailey, "SDX: A software-defined Internet exchange," *Open Networking Summit*, 2013.

[7] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *SIGCOMM*, 2011.

[8] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, 2012.

[9] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "VeriCon: Towards Verifying Controller Programs in Software-defined Networks," in *PLDI*, 2014.

[10] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *PLDI*, 2015.

[11] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *POPL*, 2016.

[12] A. Horn, A. Kheradmand, and M. R. Prasad, "Delta-net: Real-time Network Verification Using Atoms," in *NSDI*, 2017.

[13] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in *NSDI*, 2013.

[14] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *SIGCOMM*, 2016.

[15] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test Openflow Applications," in *NSDI*, 2012.

[16] Y. Jia, "NetSMC : A Symbolic Model Checker for Stateful Network Verification," in *NSDI*, 2020.

[17] V. Klimis, G. Parisis, and B. Reus, "Towards Model Checking Real-World Software-Defined Networks," in *CAV*, 2020.

[18] R. Majumdar, S. Deep Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in *FMCAD*, 2014.

[19] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang, and Q. Wang, "A survey on network verification and testing with formal methods: Approaches and challenges," *IEEE Surveys & Tutorials*, 2019.

[20] M. Al-Fares, S. Radhakrishnan, and B. Raghavan, "Hedera: Dynamic Flow Scheduling for Data Center Networks." in *NSDI*, 2010.

[21] P. Ji, Z. Ge, J. Kurose, and D. Towsley, "A comparison of hard-state and soft-state signaling protocols," *IEEE/ACM Transactions on Networking*, 2007.

[22] V. R. Pratt, "Semantical considerations on Floyd-Hoare logic," in *FOCS*, 1976.

[23] A. Pnueli, J. Xu, and L. Zuck, "Liveness with $(0, 1, \infty)$-counter abstraction," in *CAV*, 2002.

[24] D. Peled, "All from one, one for all: on model checking using representatives," in *CAV*, 1993.

[25] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "Developing UPPAAL over 15 years," *Software: Practice and Experience*, 2011.

APPENDIX

A Proofs

Lemma 1 (SAFENESS) For transition system $\mathcal{M}_{(\lambda,\text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$ and a formula $\varphi \in \text{LTL}_{\setminus\{\bigcirc\}}$, $\alpha = \mathit{fsync}(sw, r, cs)$ is safe iff the following two conditions are satisfied:

Independence CP is not order-sensitive

Invisibility `if` $Q(q)$ in AP occurs in φ, `then` α is φ-invariant

Proof. To show safety we need to show two properties: *independence* (action is independent of any other action) and *invisibility* w.r.t. the context, in particular controller program, topology function and formula φ.

Independence: Recall that two actions α and $\beta \neq \alpha$ are independent iff for any state s such that $\alpha \in A(s)$ and $\beta \in A(s)$:

(1) $\alpha \in A(\beta(s))$ and $\beta \in A(\alpha(s))$
(2) $\alpha(\beta(s)) = \beta(\alpha(s))$

(1) It can be easily checked that no instance of safe actions $\mathit{fsync}(\cdot)$ disables any other action, nor is any safe $\mathit{fsync}(\cdot)$ disabled by any other action, so the first condition of independence holds.

(2) For any safe $\alpha = \mathit{fsync}(\cdot)$ and any other action β we can assume already that they meet Condition (1). To show that any interleaving with any action $\beta \neq \alpha$ leads to the same state, we observe that

- if β is not an *fsync, ctrl* or *bsync* action, then the mutations of queues by these actions do not interfere with each other.
- The interesting cases occur when β is in $\{\mathit{fsync}(\cdot), \mathit{ctrl}(\cdot), \mathit{bsync}(\cdot)\}$. From the first condition we know that CP is not order-sensitive, which implies that α and β are independent. Order-insensitivity is a relatively strong condition but it ensures correctness of the lemma and thus partial order reduction.[5] Thus any interleaving of α and β leads to the same state.

Invisibility: $\alpha = \mathit{fsync}(sw, r, cs)$ may only affect *frq*, $sw'.fq$, $sw'.cq$ (for some switches sw'), and the control state cs. We know by definition of our Specification Language that an atomic proposition cannot refer to *frq* or any fq, cq. In case the control state changes, α is invisible to φ because of the second condition (*Invisibility*) of Lemma 1.

□

Theorem 1 (FLOW-REMOVED EQUIVALENCE) Given a property $\varphi \in \text{LTL}_{\setminus\{\bigcirc\}}$, it holds that $\mathcal{M}^{fr}_{(\lambda,\text{CP})}$ satisfies φ iff $\mathcal{M}_{(\lambda,\text{CP})}$ satisfies φ.

Proof. If $\mathit{ample}(s)$ satisfies the following conditions:

C1 (Non)emptiness condition: $\varnothing \neq \mathit{ample}(s) \subseteq A(s)$.

C2 Dependency condition: Let $s \xrightarrow{\alpha_1} s_1 \ldots \xrightarrow{\alpha_n} s_n \xrightarrow{\beta} t$ be a run in $\mathcal{M}$. If $\beta \in A \setminus \mathit{ample}(s)$ depends on $\mathit{ample}(s)$, then $\alpha_i \in \mathit{ample}(s)$ for some $0 < i \leqslant n$, which means that in every path fragment of $\mathcal{M}$, β cannot appear before some transition from $\mathit{ample}(s)$ is executed.

[5] Generalisations by a more clever analysis of the controller program are a future research topic.

C3 Invisibility condition: If $ample(s) \neq A(s)$ (i.e., state s is not fully expanded), then every $\alpha \in ample(s)$ is invisible.

C4 Every cycle in $\mathcal{M}^{fr}$ contains a fully expanded state s (i.e. $ample(s) = A(s)$).

then for each path in $\mathcal{M}$ there exists a stutter-trace equivalent path in $\mathcal{M}^{fr}$, and vice-versa, denoted $\mathcal{M} \overset{\text{st}}{\equiv} \mathcal{M}^{fr}$ – as we now show.

C1 The (non)emptiness condition is trivial since by definition of $ample(s)$ it follows that $ample(s) = \varnothing$ iff $A(s) = \varnothing$.

C2 By assumption $\beta \in A \backslash ample(s)$ depends on $ample(s)$. But with our definition of $ample(s)$ this is impossible as all actions in $ample(s)$ are safe and by definition independent of all other actions.

C3 The validity of the invisibility condition is by definition of $ample$ and safe actions.

C4 We now show that every cycle in $\mathcal{M}^{fr}_{(\lambda,\text{CP})}$ contains a fully expanded state s, i.e. a state s such that $ample(s) = A(s)$. By definition of $ample(s)$ it is equivalent to show that there is no cycle in $\mathcal{M}^{fr}_{(\lambda,\text{CP})}$ consisting of safe actions only. We show this by contradiction, assuming such a cycle of only safe actions exists.
Distinguish two cases.

Case 1 A sequence of safe actions of same type. Let ρ an execution of $\mathcal{M}^{fr}_{(\lambda,\text{CP})}$ which consists of only one type of *fsync*-actions: $\rho = s_1 \xrightarrow{fsync(sw_1,r_1,cs_1)}_{fr} s_2 \xrightarrow{fsync(sw_2,r_2,cs_2)}_{fr} \ldots s_{i-1} \xrightarrow{fsync(sw_{i-1}r_{i-1},,cs_{i-1})}_{fr} s_i$. Suppose ρ is a cycle. According to the *fsync* semantics, for each transition $s \xrightarrow{fsync(sw,r,cs)}_{fr} s'$, where $s = (\pi, \delta, \gamma)$, $s' = (\pi', \delta', \gamma')$, it holds that $\gamma'.frq = \gamma.frq \backslash \{r\}$ as we use sets to represent *frq* buffers. Hence, for the execution ρ it holds $\gamma_i.frq = \gamma_1.frq \backslash \{r_1, r_2, \ldots r_{i-1}\}$ which implies that $s_1 \neq s_i$. Contradiction.

Case 2 A sequence of different safe actions. Suppose there exists a cycle with mixed safe actions starting in s_1 and ending in s_i. Distinguish the following cases.

i) There exists at least a *fsync* action in the cycle. According to the effects of safe transitions, the *fsync* action will switch to a state with smaller *frq*. It is important here that no action of other type than *fsync* accesses *frq*. This implies that $s_1 \neq s_i$. Contradiction.

ii) No *fsync* action in the cycle. This is already established in [17].

□

Chapter 5

Conclusions and Future Work

Networks are widely acknowledged to be notoriously difficult to verify because of the intricacy involved in reasoning about concurrency. To make significant headway, networking needs general, lightweight, reusable and robust abstractions that can be reasoned about as expediently as possible. Having chosen as modelling mechanism for concurrency a communication which is based on shared-state (through interleavings), we began by developing an eminently expressive and optimised OpenFlow/SDN model that can be checked efficiently to find (or verify the absence of) latent real-world bugs. We showed that the abstractions are provably correct, preserving the initial verification promises, and demonstrated their prominence as contrasted with the state-of-the-art in terms of expressivity and performance/scalability. We then presented some enhancements to the baseline model, which, by embedding richer semantics, allow capturing aspects of flow removals caused by timeouts, complying, thus, with the OpenFlow specification. Though the final model is moderately heavyweight, we proposed additional lightweight optimisations in order to offset the additional overhead that the complex semantics usually brings along. The optimisations explore different trade-offs in time and space.

The major upshot of all the aforementioned efforts is a general framework for establishing correctness of OpenFlow-based Software-Defined Networks with mathematical rigour in a timely manner.

5.1 Future Directions

This section looks at some of the promising future directions.

Automatic discovery of equivalence classes In our ongoing research, we are investigating new techniques that make model checking SDNs faster and cheaper. A useful

direction for future work is to explore automatic computation of equivalence classes of packets that cover all visible behaviours (where we still computed semi-manually).

Exploring the benefits to using several model checkers Although we demonstrated our approach concretely using the UPPAAL model checker, it applies quite broadly; our modular abstractions can be compiled into any other type of static analysis which can also benefit from them. In this regard, comparing the performance of our abstractions using them on other state of the art model checkers and choosing the right one for a particular problem domain, is another interesting research direction that is worth investigating.

Liveness Yet, another thread in our ongoing and future work focuses on extending our formal model to provide liveness (on top of safety) promises in software-defined networks. The liveness problem, however, tend to be harder than the reachability-based safety. The reason for this is fairly straightforward: liveness properties constraint infinite behaviours and reasoning about all infinite paths to establishing liveness entails proving that there are no unfair cycles within the runs of the abstract transition system, which is computationally expensive and, in some cases, undecidable. Thus, unfair situations, like premature termination (deadlock) and starvation, may jeopardise the liveness of executions. In order to rule out such unfair situations and verify liveness in SDNs, we are taking account of fairness assumptions by exploring different traps and patterns.

More expressivity As not every OpenFlow message is incorporated in our model, such as `Features Request/Reply`, `Get Config Request/Reply`, `Set Config`, `Stats Request/Reply`, etc, future work could consider modelling new actions, allowing for more expressive models.

Verifying compliance of state changes in real time This work proposed techniques for what one might reasonably call static analysis of a Software-Defined Network. A promising direction for future work is verifying 'dynamically' properties by capturing likely changes of the forwarding state that may happen over time, either due to targeted configurations by operators or in-network degradations. This could be achieved through an incremental model checking by getting the network devices to expose incremental changes to their state to MoCS either via "pushed" SNMP traps or polling by MoCS.

Higher level specification language While our specification language leaves no space as to its interpretation, we want to explore injecting assertions (correctness properties) at higher levels of abstraction with a more intuitive notation, all while maintaining the uncompromising mathematical rigour. This will allow the intended behaviour of SDNs to be easily specified even by non-computer scientists.

Automated modelling MoCS is a man-made model from the OpenFlow specifications.

As SDN controller programs become increasingly complex, algorithms adept in automatically building more expressive models are needed. Verification of models of 'real-world' concurrent behaviour in SDNs which are based on model learning, is another unexplored and challenging area of research which can be highly effective in subduing complexity and fostering scalability.

5.2 A final remark

As a final remark, the immense complexity of Software-Defined Networks and Controller Programs is necessitating ever more elegant and powerful abstractions which will ameliorate the state space explosion, but at the same time being sufficiently expressive to capture a wide range of complex problems in a natural way. In exploring this thesis, the aim has been for the framework to be powerful, yet both simple and intuitive. The hope is that this work will mark a turning point in reasoning about networks formally and will inspire further research directed at extending our arguments.

Extended Bibliography

Note: Here is a comprehensive list of resources that were consulted during the writing of the thesis.

Cisco Systems Inc. Spanning tree protocol problems and related design considerations. http://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/10556-16.html.

Floodlight OpenFlow Controller. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/.

HASSEL-C: An optimized version of the header space library written in C. https://bitbucket.org/peymank/hassel-public/.

ITU-T Y.3300: Framework of software-defined networking. https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-Y.3300-201406-I!!PDF-E&type=items.

Mininet: An Instant Virtual Network on your Laptop (or other PC). http://mininet.org.

Nicira- It's time to virtualize the network. http://www.netfos.com.tw/PDF/Nicira/ItisTimeToVirtualizetheNetworkWhitePaper.pdf.

ns-3: Openflow switch support. https://www.nsnam.org/docs/release/3.13/models/html/openflow-switch.html.

Open Networking Foundation. https://www.opennetworking.org/.

OpenContrail. https://github.com/tungstenfabric/tf-controller.

OpenDaylight: An open source SDN controller platform. http://www.opendaylight.org/.

Opening designs for 6-pack and Wedge 100. https://code.facebook.com/posts/203733993317833/opening-designs-for-6-pack-and-wedge-100/.

POX OpenFlow controller. https://github.com/noxrepo/pox.

Ryu Controller. https://ryu-sdn.org/.

The Frenetic Research Project. http://www.frenetic-lang.org.

The LLVM Compiler Infrastructure. http://llvm.org/.

SDN Migration Considerations and Use Cases, 2014. https://opennetworking.org/wp-content/uploads/2014/10/sb-sdn-migration-use-cases.pdf.

VMware NSX Customer Story: Colt Decreases Data Center Networking Complexity, 2014. https://blogs.vmware.com/networkvirtualization/2014/08/vmware-nsx-customer-story-colt-decreases-data-center-networking-complexity.html/.

Fides Aarts and Frits Vaandrager. Learning I/O automata. In *CONCUR*, 2010.

Riadh Ben Abdallah, Tanguy Risset, Ian F. Akyildiz, et al. SoftRAN: Software defined radio access network. *IEEE Communications Magazine*, 2014.

Ian F. Akyildiz, Pu Wang, and Shih Chun Lin. SoftAir: A software defined networking architecture for 5G wireless systems. *Computer Networks*, 2015.

Mohammad Al-Fares, Sivasankar Radhakrishnan, and Barath Raghavan. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

Ehab Al-Shaer, Will Marrero, Adel El-Atawy, and Khalid ElBadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *ICNP*.

Elvira Albert, Miguel Gómez-Zamalloa, Albert Rubio, Matteo Sammartino, and Alexandra Silva. SDN-Actors: Modeling and verification of SDN programs. In *FM*, 2018.

Hassan Ali-Ahmad, Claudio Cicconetti, Antonio De La Oliva, et al. An SDN-based network architecture for extremely dense wireless networks. In *SDN4FNS*

Workshop on Software Defined Networks for Future Networks and Services, 2013.

Mohammad Alizadeh, Albert Greenberg, and Da Maltz. DCTCP: Efficient packet transport for the commoditized data center. *SIGCOMM*, 2010.

Mohammad Alizadeh, Navindra Yadav, George Varghese, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, and Rong Pan. CONGA. In *Proceedings of ACM SIGCOMM*, 2014.

Mohammad Alizadeh, Shuang Yang, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Deconstructing datacenter packet transport. *Proceedings of the 11th ACM Workshop on Hot Topics in Networks - HotNets-XI*, pages 133–138, 2012.

Bowen Alpern and Fred B. Schneider. Defining liveness. *IPL*, 1985.

Rajeev Alur and David Dill. Automata for modeling real-time systems. *Automata, languages and programming*, 443(443):322–335, 1990.

Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

Rajeev Alur, Tomás Feder, and Thomas a. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–203, 1994.

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetkAT: Semantic foundations for networks. In *POPL*, 2014.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 1987.

Muhammad Bilal Anwer, Murtaza Motiwala, Muhammad Mukarram Bin Tariq, and Nick Feamster. Switchblade: a platform for rapid deployment of network protocols on programmable hardware. *ACM SIGCOMM Computer*, pages 183–194, 2010.

Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller Synthesis For Timed Automata. *Proceedings of the IFAC Symposium on System Structure and Control*, pages 469–474, 1998.

Siamak Azodolmolky, Reza Nejabati, Eduard Escalona, Ramanujam Jayakumar, Nikolaos Efstathiou, and Dimitra Simeonidou. Integrated OpenFlow–GMPLS control plane: an overlay model for software defined packet over optical networks. *Optics Express*, 2011.

Yuval Bachar (Facebook) and Adam Simpkins (Facebook). Introducing "Wedge" and "FBOSS," the next steps toward a disaggregated network. https://code.facebook.com/posts/681382905244727/introducing-wedge-and-fboss-the-next-steps-toward-a-disaggregated-network/.

Christel Baier and Joost-Pieter Katoen. *Principles Of Model Checking*. 2008.

Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI*, pages 282–293, 2013.

Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *PLDI*, 2014.

Manu Bansal, Jeffrey Mehlman, Sachin Katti, and Philip Levis. OpenRadio: A Programmable Wireless Dataplane. In *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN*, 2012.

Ryan Beckett, Michael Greenberg, and David Walker. Temporal NetKAT. In *PLDI*, 2016.

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *SIGCOMM*, 2017.

Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 years. *Software: Practice and Experience*, 2011.

T Benson, A Anand, A Akella, and M Zhang. MicroTE: fine grained traffic engineering for data centers. In *ACM CoNEXT*, 2011.

Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, and Bob Lantz. ONOS: towards an open, distributed SDN OS. *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN*, pages 1–6, 2014.

Carlos J. Bernardos, Antonio De La Oliva, Pablo Serrano, Albert Banchs, Luis M. Contreras, Hao Jin, and Juan Carlos Zúñiga. An architecture for software defined wireless networking. *IEEE Wireless Communications*, 2014.

Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. *Advances in Computers*, 58(99):117–148, 2003.

Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*, volume 185. 2009.

Brad Bingham, Jesse Bingham, Flavio M. De Paula, John Erickson, Gaurav Singh, and Mark Reitblatt. Industrial strength distributed explicit state model checking. In *PDMC*, 2010.

Pat Bosshart (Barefoot Networks), Dan Daly (Intel), Glen Gibb (Barefoot Networks), Nick McKeown (Stanford University), et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 2014.

Rodrigo Braga, Edjard Mota, and Alexandre Passito. Lightweight DDoS flooding attack detection using NOX/OpenFlow. In *LCN*, 2010.

Sebastian Brandt, Klaus Tycho Foerster, and Roger Wattenhofer. Augmenting flows for the consistent migration of multi-commodity single-destination flows in SDNs. *Pervasive and Mobile Computing*, 2017.

Sebastian Brandt, Klaus Tycho Förster, and Roger Wattenhofer. On consistent migration of flows in SDNs. In *IEEE INFOCOM*, 2016.

M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1-2):115–131, 1988.

Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

Randal E. Bryant and Carl-Johan H. Seger. Formal verification of digital circuits using symbolic ternary system models. In *Computer-Aided Verification (CAV)*, pages 33–43, 1991.

Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM, Magazine*, 2013.

Zheng Cai, Alan Cox, and Eugene T. S. Ng. Maestro: A System for Scalable OpenFlow Control. *Cs.Rice.Edu*, page 10, 2011. https://www.cs.rice.edu/~eugeneng/papers/TR10-11.pdf.

Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE Way to Test Openflow Applications. In *NSDI*, 2012.

S.K. Card, G.G. Robertson, and J.D. Mackinlay. The Information Visualizer: An Information Workspace, 1991.

Francisco Carpio, Anna Engelmann, and Admela Jukan. DiffFlow: Differentiating short and long flows for load balancing in data center networks. In *IEEE Global Communications Conference, GLOBECOM*, 2016.

Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. *SIGCOMM*, pages 1–12, 2007.

Min Cheng Chan, Chien Chen, Jun Xian Huang, Ted Kuo, Li Hsing Yen, and Chien Chao Tseng. OpenNet: A simulator for software-defined wireless local area network. In *IEEE Wireless Communications and Networking Conference, WCNC*, pages 3332–3336, 2014.

M. Channegowda, R. Nejabati, M. Rashidi Fard, et al. Experimental demonstration of an OpenFlow based software-defined optical network employing packet, fixed and flexible DWDM grid technologies on an international multi-domain testbed. *Optics Express*, 2013.

Tao Chen, Honggang Zhang, Xianfu Chen, and Olav Tirkkonen. SoftMobile: Control evolution for future heterogeneous mobile networks. *IEEE Wireless Communications*, 2014.

Stuart Cheshire. Latency and the Quest for Interactivity, 1996.

Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM Computer Communication Review*, 41(4):98, 2011.

Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, et al. Nusmv 2: An open-source tool for symbolic model checking. In *Computer Aided Verification (CAV)*, pages 359–364, 2002.

D. Clark. The design philosophy of the DARPA internet protocols. *ACM SIGCOMM Computer Communication Review*, 1988.

E. Clarke, K. McMillan, Sérgio Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification (CAV)*, volume 1102, pages 419–422. 1996.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

E M Clarke, J M Wing, R Alur, R Cleaveland, D Dill, A Emerson, S Garland, and Others. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

Edmund Clarke and Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. *Logic of Programs*, 1981.

Edmund M. Clarke. *The Birth of Model Checking*, pages 1–26. 2008.

Edmund M. Clarke, Orna Grumberg, and Doron A Peled. Model Checking, 1999.

Gerald Combs. Wireshark, 2019. https://www.wireshark.org/.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 1977.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL*, pages 269–282, 1979.

Andrew R. Curtis, Wonho Kim, and Praveen Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *IEEE INFOCOM*, 2011.

Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: scaling flow management for high-performance networks. *SIGCOMM*, 2011.

Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos Made Switch-y. *ACM SIGCOMM Computer Communication Review*, 2016.

Alexandre David, Kim G. Larsen, Axel Legay, et al. Statistical model checking for networks of priced timed automata. In *Formal Modeling and Analysis of Timed Systems - FORMATS*, 2011.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - TACAS*, 2008.

Peter Dely, Andreas Kassler, and Nico Bayer. OpenFlow for wireless mesh networks. In *International Conference on Computer Communications and Networks, ICCCN*, 2011.

David L. Dill. The Murϕ verification system. In *CAV*, 1996.

Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. *Communications of the ACM*, 2015.

Mihai Dobrescu, Norbert Egi, Katerina Argyraki, et al. RouteBricks: exploiting parallelism to scale software routers. *ACM SIGOPS 22nd symposium on Operating systems principles SE - SOSP*, pages 15–28, 2009.

Szymon Dudycz, Arne Ludwig, and Stefan Schmid. Can't touch this: Consistent network updates for multiple policies. In *DSN*, 2016.

Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing - SAT*. 2010.

Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDN Racer: Concurrency analysis for software-defined networks. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-Wide configuration synthesis. In *Computer Aided Verification (CAV)*, 2017.

E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time Temporal Logic. *Journal of the ACM (JACM)*, 1986.

D. Erickson. A demonstration of virtual machine mobility in an OpenFlow network. *ACM SIGCOMM, (Best Demo Award)*, 2008.

David Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN*, page 13, 2013.

Nathan Farrington, George Porter, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Hunting mice with microsecond circuit switches. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks - HotNets-XI*, pages 115–120, 2012.

Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *OSDI*, 2016.

Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *NSDI*, 2016.

Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, Ron Hutchins, Dave Levin, and Josh Bailey. SDX: A software-defined Internet exchange. *Open Networking Summit*, 2013.

Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN. *SIGCOMM Computer Communication Review*, 2014.

Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 1979.

Klaus Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. Survey of Consistent Software-Defined Network Updates. *IEEE Communications Surveys and Tutorials*, 2019.

Ari Fogel, Stanley Fung, Luis Pedrosa, et al. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 469–483, 2015.

Klaus Tycho Forster, Ratul Mahajan, and Roger Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *IFIP Networking 2016*, 2016.

Klaus Tycho Förster and Roger Wattenhofer. The power of two in consistent network updates: Hard loop freedom, easy flow migration. In *ICCCN*, 2016.

Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming - (ICFP)*, volume 46, page 279, 2011.

Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic NetKAT. In *ESOP*, 2016.

Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *POPL*, 2015.

Aaron Gember, Anand Krishnamurthy, Saul St. John, et al. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud, 2014. arxiv:1305.0209 [cs.NI].

Glenn Gibb, John W. Lockwood, Jad Naous, Paul Hartke, and Nick McKeown. NetFPGA - An open platform for teaching how to build gigabit-rate network switches and routers. *IEEE Transactions on Education*, 51(3):364–369, 2008.

Nati Shalom (GigaSpaces). Amazon found every 100ms of latency cost them 1% in sales, 2008. https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/.

Global Environment for Network Innovations (GENI). OpenFlow Firewall Assignment. https://groups.geni.net/geni/wiki/GENIEducation/

SampleAssignments/OpenFlowFirewallAssignment/ExerciseLayout/Execute.

David P. Gluch, Santiago Comella-Dorda, John Hudak, Grace Lewis, John Walker, Charles B. Weinstock, and David Zubrow. Model-Based Verification: An Engineering Practice. Technical report, Carnegie Mellon University, 2002.

Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods. In *Computer Aided Verification (CAV)*, pages 438–449, 1993.

Kate Greene. TR10: Software-defined networking. *MIT Technology Review*, 2009. http://www2.technologyreview.com/article/412194/tr10-software-defined-networking/.

Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking*, 2002.

Jun Gu, P W Purdom, Jhon Franco, and B W Wah. Algorithms for the satisfiability (sat) problem. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 00, pages 19–152, 1996.

Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM Computer Communication Review*, 38(3):105, 2008.

Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. *ACM SIGPLAN Notices*, 48(6):483, 2013.

GUROBI Optimization Inc. Gurobi Optimizer reference manual. Technical report, 2018. http://www.gurobi.com.

Gustavo J.A.M Carneiro. ns-3:Network Simulator 3 Tutorial. https://www.nsnam.org/tutorials/NS-3-LABMEETING-1.pdf.

Sangjin Han, Keon Jang, Kyoungsoo Park, and Sue Moon. PacketShader: a GPU-Accelerated Software Router. In *ACM SIGCOMM*, 2010.

Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emula-

tion. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies - CoNEXT*, 2012.

Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN*, 2012.

Nikhil Handigol, Srini Seetharaman, Mario Flajslik, and Aaron Gember. Aster * x : Load-Balancing Web Traffic over Wide-Area Networks. *Deutsche Telekom R&D*, 2010.

Nikhil Handigol, Srinivasan Seetharaman, Mario Flajslik, Nick McKeown, and Ramesh Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow. *SIGCOMM*, 2009.

R. Handigol, N., Flajslik, M., Seetharaman, Johari. Aster*x: load-balancing as a network primitive. *In Proceedings of Architectural Concerns in Large Datacenters (ACLD)*, 2010.

Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM*, 2017.

Soheil Hassas Yeganeh, Yashar Ganjali, Soheil Hassas Yeganeh, and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24, 2012.

Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *STTT*, 2000.

Brandon Heller, James McCauley, Kyriakos Zarifis, Peyman Kazemian, Colin Scott, Nick McKeown, Scott Shenker, Andreas Wundsam, Hongyi Zeng, Sam Whitlock, Vimalkumar Jeyakumar, and Nikhil Handigol. Leveraging SDN layering to systematically troubleshoot networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN*, page 37, 2013.

Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree : Saving Energy in Data Center Networks. *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, pages 17–17, 2010.

T A Henzinger, X Nicollin, J Sifakis, and S Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 1997.

Gerard J. Holzmann and Doron Peled. An Improvement in Formal Verification. In *FORTE*, 1994.

Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM*, page 127, 2012.

Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition, 2001.

Alex Horn, Ali Kheradmand, and Mukul R. Prasad. Delta-net: Real-time Network Verification Using Atoms. In *NSDI*, 2017.

Hongxin Hu, Gail-Joon Ahn, Wonkyu Han, and Ziming Zhao. Towards a Reliable SDN Firewall. In *ONS*, 2014.

Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. FlowGuard: Building Robust Firewalls for Software-Defined Networks. *HotSDN*, 2014.

Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*, volume 16. 2006.

International Telecommunication Union. ITU-T Recommendation G. 1010: End-user multimedia QoS categories (Quality of service and performance). *International Telecommunications Union*, 1010, 2001.

ITU-T. G.114 One-way transmission time. *SERIES G: TRANSMISSION SYSTEMS AND MEDIA, DIGITAL SYSTEMS AND NETWORKS International telephone connections and circuits – General Recommendations on the transmission quality for an entire international telephone connection*, pages 1–20, 2003.

Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 2002.

Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. OpenFlow random host mutation: Transparent moving target defense using software defined networking. In *HotSDN*, 2012.

Sushant Jain, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, Amin Vahdat, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, and Junlan Zhou. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

Michael Jarschel, Thomas Zinner, Tobias Hossfeld, Phuoc Tran-Gia, and Wolfgang Kellerer. Interfaces, attributes, and use cases: A compass for SDN. *IEEE Communications Magazine*, 52(6):210–217, 2014.

Vimalkumar Jeyakumar, David Mazi, and Changhoon Kim. EyeQ : Practical Network Performance Isolation for the Multi-tenant Cloud. *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, page 8, 2012.

Ping Ji, Zihui Ge, Jim Kurose, and Don Towsley. A comparison of hard-state and soft-state signaling protocols. *IEEE/ACM Transactions on Networking*, 2007.

Yifei Jia. NetSMC : A Symbolic Model Checker for Stateful Network Verification. In *NSDI*, 2020.

Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *the ninth ACM conference on Emerging networking experiments and technologies*, 2013.

Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. *ACM SIGCOMM CCR*, 2015.

Simon Jouet, Colin Perkins, and Dimitrios Pezaros. OTCP: SDN-managed congestion control for data center networks. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2016.

Tobias Kappé, Paul Brunet, Alexandra Silva, and Fabio Zanasi. Concurrent kleene algebra: Free model and completeness. In *ESOP*, 2018.

Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM Symposium on SDN Research (SOSR)*, 2016.

Naga Praveen Katta, Jennifer Rexford, and David Walker. Logic Programming for Software-Defined Networks. *Workshop on Cross-Model Design and Validation (XLDI), ACM*, 2012.

Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking - HotSDN*, 2013.

Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 1992.

Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*, 2013.

Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI*, 2013.

Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Building distributed systems using Mace. In *IEEE - 9th International Conference on Peer-to-Peer Computing (P2P)*, pages 91–92, 2009.

Charles Killian, Charles Killian, James W Anderson, James W Anderson, Ranjit Jhala, Ranjit Jhala, Amin Vahdat, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.

Charles Edwin Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin M Vahdat. Mace. *ACM SIGPLAN Notices*, 42(6):179, 2007.

Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable Dynamic Network Control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 59–72, 2015.

Vasileios Klimis, George Parisis, and Bernhard Reus. Model Checking Software-Defined Networks with Flow Entries that Time Out (version with appendix). *arXiv: 2008.06149 [cs.NI]*, 2020.

Vasileios Klimis, George Parisis, and Bernhard Reus. Towards Model Checking Real-World Software-Defined Networks. In *CAV*, 2020.

Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE*, 2011.

Masayoshi Kobayashi, Srini Seetharaman, Guru Parulkar, Guido Appenzeller, Joseph Little, Johan Van Reijendam, Paul Weissmann, and Nick McKeown. Maturing of OpenFlow and Software-defined Networking through deployments. *Computer Networks*, 2014.

Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

Teemu Koponen, Martin Casado, Natasha Gude, et al. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–6, 2010.

Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

Dexter Kozen. Kleene Algebra with Tests. *TOPLAS*, 1997.

Diego Kreutz, Fernando M V Ramos, Paulo Esteves Verissimo, et al. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

Saul Kripke. Semantical Considerations on Modal Logic, 1963.

Ian Ku, You Lu, Mario Gerla, Rafael L. Gomes, Francesco Ongaro, and Eduardo Cerqueira. Towards software-defined VANET: Architecture and services. In *13th Annual Mediterranean Ad Hoc Networking Workshop, MED-HOC-NET*, 2014.

Rui Kubo, Tomonori Fujita, Yuji Agawa, and Hikaru Suzuki. Ryu SDN framework-open-source SDN platform software. *NTT Technical Review*, 12(8), 2014.

Marta Kwiatkowska. Quantitative verification: Models, Techniques and Tools. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering - ESEC-FSE*, 2007.

Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. *Formal methods for performance ...*, 2007.

Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: probabilistic model checking for performance and reliability analysis. *SIGMETRICS Perform. Eval. Rev.*, 36(4):40–45, 2009.

Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification (CAV)*, pages 585–591, 2011.

Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *ACM SIGCOMM*, 2016.

T V Lakshman and D Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *Computer Communication Review*, 28(4):203–214, 1998.

Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 1978.

Leslie Lamport. What Good is Temporal Logic?, 1983.

Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets*, pages 1–6, 2010.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, 2001.

Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical Model Checking: An Overview. In *International Conference on Runtime Verification*, 2010.

D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *ICALP*, 1981.

Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. In *IEEE Transactions on Computers*, 1985.

Li Erran Li, Z. Morley Mao, and Jennifer Rexford. Toward software-defined cellular networks. In *Proceedings - European Workshop on Software Defined Networks (EWSDN)*, 2012.

Yahui Li, Xia Yin, Zhiliang Wang, Jiangyuan Yao, Xingang Shi, Jianping Wu, Han Zhang, and Qing Wang. A survey on network verification and testing with formal methods: Approaches and challenges. *IEEE Surveys & Tutorials*, 2019.

Yu Li and Deng Pan. OpenFlow based Load Balancing for Fat-Tree Networks with Multipath Support. In *Proc. 12th IEEE International Conference on Communications (ICC)*, 2013.

Greg Linden. Marissa Mayer at Web 2.0, 2006.

Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating data center networks with zero loss. In *SIGCOMM*, 2013.

Jiajia Liu, Shangwei Zhang, Nei Kato, Hirotaka Ujikawa, and Kenichi Suzuki. Device-to-device communications for enhancing quality of experience in software defined multi-tier LTE-A networks. *IEEE Network*, 2015.

Lei Liu, Takehiro Tsuritani, Itsuro Morita, Hongxiang Guo, and Jian Wu. Experimental validation and performance evaluation of OpenFlow-based wavelength path control in transparent optical networks. *Optics Express*, 2011.

Lei Liu, Dongxu Zhang, Takehiro Tsuritani, Ricard Vilalta, Ramon Casellas, Linfeng Hong, Itsuro Morita, Hongxiang Guo, Jian Wu, Ricardo Martinez, and Raül Munoz. Field trial of an openflow-based unified control plane for multilayer multigranularity optical switching networks. *Journal of Lightwave Technology*, 2013.

Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.

Arne Ludwig, Szymon Dudycz, Matthias Rost, and Stefan Schmid. Transiently secure network updates. In *SIGMETRICS/ Performance*, 2016.

Long Luo, Hongfang Yu, Shouxi Luo, and Mingui Zhang. Fast lossless traffic migration for SDN updates. In *IEEE International Conference on Communications*, 2015.

Tie Luo, Hwee Pink Tan, and Tony Q S Quek. Sensor openflow: Enabling software-defined wireless sensor networks. *IEEE Communications Letters*, 2012.

Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *FMCAD*, 2014.

David Makinson. *Sets, logic and maths for computing*. 2008.

Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient synthesis of network updates. In *PLDI*, 2015.

Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cěrnyá. Event-driven network programming. In *PLDI*, 2016.

Nick McKeown. Mind the Gap. In *SIGCOMM Keynote*, 2014.

Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

Kenneth L. McMillan. Symbolic Model Checking. In *Symbolic Model Checking*, pages 25–60. 1993.

Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. OpenDaylight: Towards a model-driven SDN controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014, (WoWMoM)*, 2014.

Robert B. Miller. Response time in man-computer conversational transactions. In *AFIPS*, page 267, 1968.

Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever, and Martin Vechev. SDNRacer: Detecting Concurrency Violations in Software-defined Networks. *Sosr*, pages 22:1–22:7, 2015.

John C. Mitchell, Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, and Scott Shenker. Practical declarative network management. 2009.

Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL*, page 217, 2012.

Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 1–14, 2013.

R. Mortier, T. Rodden, T. Lodge, D. McAuley, C. Rotsos, A. W. Moore, A. Koliousis, and J. Sventek. Control and understanding: Owning your home network. In *2012 4th International Conference on Communication Systems and Networks, COMSNETS 2012*, 2012.

P. Godefroid N. Lopes, N. Bjørner and G. Varghese. Network Verification in the Light of Program Verification. 2013.

Sriram Natarajan, Anantha Ramaiah, and Mayan Mathen. A Software defined Cloud-Gateway automation system using OpenFlow. In *IEEE 2nd International Conference on Cloud Networking (CloudNet)*, pages 219–226, 2013.

Ak Nayak and Alex Reimers. Resonance: dynamic access control for enterprise networks. *Wren*, pages 11–18, 2009.

Tim Nelson, Andrew D Ferguson, Michael J G Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-defined Networks. In *NSDI*.

Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, Analyzable Controller Programming Tim. *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN*, 2013.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. 2013.

Oliver Niese. *An integrated approach to testing complex systems*. PhD thesis, University of Dortmund, 2003.

Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless Datacenter Load-balancing with Beamer. In *NSDI*, 2018.

ONF. Software-defined networking: The new norm for networks. *White Paper*, 2012.

ONF. SDN Architecture Overview. *Technical Report*, 2013.

Open Networking Foundation. SDN Architecture. *Onf*, 2014.

Open Networking Foundation. OpenFlow Switch Specification 1.5.1. Technical report, 2015.

Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *Proceedings of the ACM on Programming Languages*, 2018.

Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David a Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. *SIGCOMM*, 2013.

Doron Peled. All from one, one for all: on model checking using representatives. In *CAV*, 1993.

Doron Peled. Partial order reduction: Model-checking using representatives. In *Mathematical Foundations of Computer Science 1996*, pages 93–112, 1996.

Doron Peled, Moshe Vardi, and Mihalis Yannakakis. Black Box Checking. *Journal of Automata, Languages and Combinatorics*, 2002.

Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.

Peter Peresini, Maciej Kuzniar, and Dejan Kostic. Dynamic, Fine-Grained Data Plane Monitoring with Monocle. *IEEE/ACM Transactions on Networking*, 2018.

Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devarat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. *ACM SIGCOMM*, pages 307–318, 2015.

Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. In *POPL*, 2016.

Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

Amir Pnueli, Jessie Xu, and Lenore Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *CAV*, 2002.

Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, 2005.

Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification, 2005.

Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *FOCS*, 1976.

Zafar Ayyub Qazi, Cheng Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM*, 2013.

Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *SIGCOMM*, 2012.

Abhinava Sadasivarao, Sharfuddin Syed, Ping Pan, Chris Liou, Andrew Lake, Chin Guok, and Inder Monga. Open Transport Switch: A Software Defined Networking Architecture for Transport Networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN*, 2013.

Mohammad Ali Salahuddin, Ala Al-Fuqaha, and Mohsen Guizani. Software-defined networking for rsu clouds in support of the internet of vehicles. *IEEE Internet of Things Journal*, 2015.

Brandon Schlinker, Hongyi Zeng, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, and Petr Lapukhov. Engineering Egress with Edge Fabric. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM*, pages 418–431, 2017.

Dana S. Scott. Outline of a mathematical theory of computation. *Technical Monograph PRG-2, Oxford University Computing Laboratory*, 1970.

Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. *Computer Aided Verification (CAV)*, 2004.

Divjyot Sethi, Srinivas Narayana, and Sharad Malik. Abstractions for model checking SDN controllers. In *FMCAD*, 2013.

Scott Shenker, Martin Casado, Teemu Koponen, and Nick McKeown. The future of networking, and the past of protocols. In *ONS*, 2011.

Rob Sherwood, Glen Gibb, Kk Kok-Kiong Kk Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru M Parulkar. Can the Production Network Be the Testbed? *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, M(1):365–378, 2010.

Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. *Nsdi*, pages 23–23, 2011.

Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. Rosemary: A Robust, Secure, and High-Performance Network Operating System. *ACM SIGSAC Conference on Computer and Communications Security - CCS*, pages 78–89, 2014.

Richard Skowyra, Andrei Lapets, Azer Bestavros, and Assaf Kfoury. A verification platform for SDN-enabled applications. In *IC2E*, 2014.

Anthony M. Sloane. Software Abstractions: Logic, Language, and Analysis by Daniel Jackson, The MIT Press, 2006, 366pp, ISBN 978-0262101141. *Journal of Functional Programming*, 2009.

Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Model checking invariant security properties in OpenFlow. In *IEEE*, 2013.

Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. *ACM SIGOPS Operating Systems Review*, 35(5):216, 2001.

Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring ISP Topologies With Rocketfuel. *IEEE/ACM Transactions on Networking*, 2004.

Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable symbolic execution for modern networks. In *SIGCOMM*, 2016.

D. E. Taylor, J. S. Turner, and J. W. Lockwood. Dynamic hardware plugins (DHP): Exploiting reconfigurable hardware for high-performance programmable routers. In *IEEE Open Architectures and Network Programming Proceedings (OPENARCH)*, pages 25–34, 2001.

Amin Tootoonchian and Y Ganjali. Hyperflow: a distributed control plane for openflow. *Internet Network Management Workshop / Workshop on Research on Enterprise Networking (INM/WREN)*, pages 3–3, 2010.

Alex F.R. Trajano and Marcial P. Fernandez. Two-phase load balancing of In-Memory Key-Value Storages through NFV and SDN. In *Proceedings - IEEE Symposium on Computers and Communications*, 2016.

Ramona Trestian. MiceTrap: Scalable traffic engineering of datacenter mice flows using OpenFlow. *IFIP/IEEE International Symposium on Integrated Network Management*, 2013.

Jeffrey D. Ullman. *Foundations of Computer Science*. 1992.

Frits Vaandrager. Model learning. *Communications of the ACM*, 2017.

Ronald Van Der Pol, Sander Boele, Freek Dijkstra, Artur Barczyk, Gerben Van Malenstein, Jim Hao Chen, and Joe Mambretti. Multipathing with MPTCP and open flow. In *High Performance Computing, Networking Storage and Analysis, SCC*, 2012.

Bas C. van Fraassen. *Formal Semantics and Logic*. 2016.

Moshe Y. Vardi. Automatic Verification of Probabilistic Concurrent Finite-State Systems. In *26th Annual Symposium on Foundations of Computer Science (FOCS)*, 1985.

Moshe Y. Vardi. Verification of concurrent programs: the automata-theoretic framework. *Annals of Pure and Applied Logic*, 1987.

G. Varghese. Vision for Network Design Automation and Network Verification. In *NetPL (Talk)*, 2018.

George Varghese. *Network Algorithmics*. 2005.

Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. Practical TDMA for datacenter ethernet. In *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys*, page 225, 2012.

Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 43–48, 2012.

Juan Wang, Yong Wang, Hongxin Hu, Qingxin Sun, He Shi, and Longjie Zeng. Towards a security-enhanced firewall application for openflow networks. In *Cyberspace Safety and Security*, 2013.

Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-Based Server Load Balancing Gone Wild Into the Wild : Core Ideas. *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks (Hot-ICE)*, page 12, 2011.

Shie Yuan Wang, Chih Liang Chou, and Chun Ming Yang. EstiNet openflow network simulator and emulator. *IEEE Communications Magazine*, 51(9):110–117, 2013.

Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. In *The Structure and Dynamics of Networks*. 2011.

Bernard M. Waxman. Routing of Multipoint Connections. *IEEE Journal on Selected Areas in Communications*, 1988.

I Widjaja and A Elwalid. MATE: MPLS adaptive traffic engineering. *IETF Draft*, pages 1–10, 1999.

Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. *Proc. ACM Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, pages 50–61, 2011.

Glynn Winskel and Mogens Nielsen. Models for Concurrency. *DAIMI Report Series*, 1993.

Xin Wu and Xiaowei Yang. DARD: Distributed adaptive routing for datacenter networks. In *Proceedings - International Conference on Distributed Computing Systems*, pages 32–41, 2012.

Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On static reachability analysis of IP networks. In *Proceedings - IEEE INFOCOM*, volume 3, pages 2170–2183, 2005.

Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 2016.

Mao Yang, Yong Li, Depeng Jin, Li Su, Shaowu Ma, and Lieguang Zeng. OpenRAN: A Software-defined Ran Architecture via Virtualization. In *Proceedings of the ACM SIGCOMM*, 2013.

Guang Yao, Jun Bi, and Peiyao Xiao. Source address validation solution with OpenFlow/NOX architecture. In *Proceedings - International Conference on Network Protocols, ICNP*, 2011.

Kok-Kiong Yap, Rob Sherwood, Masayoshi Kobayashi, Te-Yuan Huang, Michael Chan, Nikhil Handigol, Nick McKeown, and Guru Parulkar. Blueprint for introducing innovation into wireless mobile networks. In *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures - VISA*, 2010.

Volkan Yazici, Ulas C. Kozat, and M. Oguz Sunay. A new control plane for 5G network architecture with a case study on unified handoff, mobility, and routing management. *IEEE Communications Magazine*, 2014.

Yiannis Yiakoumis, Kok-Kiong Yap, Sachin Katti, Guru Parulkar, and Nick McKeown. Slicing home networks. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Home Networks (HomeNets)*, page 1–6, 2011.

Zuoning Yin, Matthew Caesar, and Yuanyuan Zhou. Towards Understanding Bugs in Open Source Router Software. *ACM SIGCOMM Computer Communication Review*, 40(3):35–40, 2010.

Hakan Lorens Samir Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon, 2005.

Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM*, 2017.

Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. *IEEE/ACM Transactions on Networking*, 22(2):554–566, 2014.

Hongyi Zeng, Peyman Kazemian, George Varghese, and McKeown Nick. A Survey on Network Troubleshooting. *Technical Report TR12-HPNG-061012, Stanford University*, 2012.

Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *NSDI*, 2014.

Hang Zhang, Sophie Vrzic, Gamini Senarath, Hamid Farmanbar, Jaya Rao, Chenghui Peng, and Hongcheng Zhuang. 5G wireless network: MyNET and SONAC. *IEEE Network*, 2015.

Shuyuan Zhang and Sharad Malik. SAT based verification of network data planes. In *Automated Technology for Verification and Analysis. Springer*, 2013.

R. ZHANG, S., MALIK, S., AND MCGEER. Verification of computer switching networks: An overview. In *Proceedings of the 10th international conference on Automated Technology for Verification and Analysis (ATVA)*, 2012.

Jiaqi Zheng, Hong Xu, Guihai Chen, and Haipeng Dai. Minimizing transient congestion during network update in data centers. In *ICNP*, 2016.

Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. Enforcing customizable consistency properties in software-defined networks. In *NSDI*, 2015.

Appendix A

Artifact for Paper: "Towards Model Checking Real-World Software-Defined Networks"

Artifact for Paper: "Towards Model Checking Real-World Software-Defined Networks"

Introduction

This artifact provides a self-contained virtual instance of our model-checking environment, code-named MoCS, for verifying properties of Software-Defined Networks. It contains a user-facing application, a library of SDN transition system instantiations, i.e. data-plane topologies and controller programs for all the experiments detailed in the paper (exported by UPPAAL[1] as xml files), the respective invariant properties to be checked (exported by UPPAAL as *q* files), and the back-end verification engine (http://www.uppaal.org). In the following sections, we (1) discuss the reproducibility of the experimentation included in our paper with respect to the evaluation criteria, (2) describe the submitted artifact, (3) explain the command structure and syntax conventions and (4) provide instructions for re-running the experiments presented in Section 4 of the paper.

Artifact Description

The artifact is a Ubuntu 18.04.4 Virtual Machine (VM), available for download here: https://tinyurl.com/y95qtv5k. The VM is in an OVA (Open Virtual Appliance) image format and was created with VirtualBox (https://www.virtualbox.org) on a MacBook Pro (2.5 GHz Dual-Core Intel Core i7, with 16 GB 2133 MHz LPDDR3 RAM) host. The image can be imported by most virtualisation platforms, such as VirtualBox and VMWare Fusion. Please import the boot volume (guest) into your hosted hypervisor following the vendor's documentation. Upon booting the OS, you will be logged in automatically, but should you need to perform any adjustment to the system (install, remove or change any piece of software), if prompted for the user's password when running a `sudo` command, enter `mocs`. *Note that no extra software is required for reproducing the results in the paper.* The VM is set to have 1 CPU and 4GB of RAM. As discussed below, you will need to increase the memory assigned to the virtual machine for verifying the correctness of specific SDNs. The experimentation presented in the paper has been conducted on an 18-Core iMac pro, 2.3GHz Intel Xeon W with 128GB DDR4 memory, running OSX.

Reproducibility of Results

MoCS has been experimentally evaluated in terms of (i) performance (states explored, verification throughput and memory footprint), by scaling the topology of the data-plane up, and (ii) model expressivity. We have stress-tested **MoCS** on real-world SDNs, as described in Section 4 of the paper. All the raw data collected in the experiments reported in the Experimental Evaluation section of our paper can be found in `~/Logs/`.

Completeness. With the submitted artifact one can re-run all experiments included in the paper (see below for memory-related restrictions). Currently, `mocs` is just a convenience Python script that passes to the UPPAAL back-end a specific .xml file and *q* property, which are determined by the input arguments (see section below). One needs to pre-generate these files so that verification can be executed. As a result, currently, we are not able to accept any other input.

Consistency. In Section 4.1 of the paper, we presented performance evaluation and comparison with Kuai and reported (1) verification throughput in visited states per second, (2) number of visited states and (3) required memory. The verification throughput depends on the capabilities of the host

[1] http://www.uppaal.org

onto which experiments are run, therefore the exact numbers cannot be reproduced. However, the trends reported in Figure 4 can be easily reproduced. The reported results on the number of visited states (Figures 5 and 6) and required memory can be fully reproduced. In Section 4.2 of the paper, we evaluated MoCS expressivity in comparison to Kuai, by finding bugs in SDNs that Kuai could not find. As reported in the paper, bug finding for all studied scenarios was very quick and this can be reproduced with the submitted artifact. Kuai wouldn't be able to find any of these bugs and this can be reproduced with the submitted artifact by running our own Kuai implementation. After removing the bugs from the SDN controller code, the correctness of the SDN with respect to the given properties can be verified by MoCS and this can be reproduced with our artifact.

Memory Requirements. The appliance comes pre-configured to use 4GB memory. However, in order to deal with larger verification tasks, it will be needed to increase the memory allocated to it initially. The VM will reserve all the memory you allocate to it on your host machine, so make sure there is enough spare physical memory. The table below shows the memory needed for the verification process for each setting. Memory is classified into ranges that are each assigned a different colour (see quantitative colour legend).

Expected Runtime. The lower half of the below table details the verification runtimes which are colour-coded as shown in the legend at the bottom of table. We show runtimes from an iMac pro machine with 18-Core, 2.3GHz Intel Xeon W with 128GB DDR4 memory, running OSX.

MEMORY

Controller Program	Switches	Hosts	MOCS [GB]	MOCS w/o POR [GB]	MOCS w/o any optim [GB]	Kuai [GB]
MAC learning	3	2	0.009	3.687		0.039
	4	2	0.010			0.244
	3	3	0.010	6.276		0.265
	5	2	0.016			3.056
	4	3	0.020			5.572
	6	2	0.044			49.381
	3	4	0.073			
	5	3	0.099			
	7	2	0.322			
	4	4	0.409			
	6	3	0.649			
	4	5	0.709			
	3	5	1.701			
	8	2	2.154			
	5	4	3.618			
	7	3	6.809			
	3	6	6.969			
	9	2	14.464			
	10	2				

Stateless Firewall	2	2	0.007	0.341		0.020
	3	2	0.008			0.033
	4	2	0.008			0.050
	5	2	0.009			0.083
	6	2	0.014			0.176
	7	2	0.050			0.601
	8	2	0.234			2.896
	9	2	1.352			15.595
	10	2	8.066			81.687
	11	2				
	12	2				
	13	2				
Stateful Firewall	1	2	0.009	0.073		0.014
	2	2	0.058			0.180
	3	2	5.968			24.183
	4	2				

>128GBs (>24h)	< 4GB	(4, 8] GB	(8, 16] GB	(16, 64] GB	(64, 128] GB

TIME						
MAC learning	3	2	41ms	31.59m		550ms
	4	2	399ms			13.39s
	3	3	662ms	58.05m		20.11s
	5	2	3.7s			3.63m
	4	3	8s			7.97m
	6	2	31.81s			56.16m
	3	4	53.67s			
	5	3	1.55m			
	7	2	4.53m			
	4	4	7.92m			
	6	3	13.24m			
	4	5	13.24m			
	3	5	30.73m			
	8	2	37.06m			
	5	4	1.37h			
	7	3	2.19h			
	3	6	2.4h			
	9	2	4.98h			
	10	2				

Stateless Firewall	2	2	3ms	3.58m		31ms
	3	2	27ms			92ms
	4	2	193ms			317ms
	5	2	1.45s			1.6s
	6	2	10.44s			9.69s
	7	2	1.2m			57.83s
	8	2	8.16m			5.79m
	9	2	55.58m			33.97m
	10	2	6.42h			3.29h
	11	2				
	12	2				
	13	2				

Stateful Firewall	1	2	99ms	18.38s		111ms
	2	2	24.45s			33.47s
	3	2	1.22h			2.04h
	4	2				

>24h (>128GBs)	< 1 min	(1, 10] min	(10, 60] min	> 1 hour	

Command Structure and Syntax Conventions

To launch a terminal, select either the Activities launcher in the upper-left corner of the desktop or the Show Applications icon in the lower-left corner: in the search box, enter terminal and select Terminal to open it and you will see the bash shell. To launch **MoCS**, type `mocs` along with any associated information (arguments), such as SDN controller program (CP), whether the model is an optimised one or not, and the topology information of the forwarding plane. `mocs` is a python front-end-script (located in `~/bin/`) which is executable from anywhere on the system by just typing in its name, without having to include the full path. UPPAAL (located in `~/uppaal64-4.1.19`) is the back-end engine of **MoCS**.
The general format of the `mocs` command is:

```
mocs <cp> <sr> <por> <switches> <hosts>
```

The table below describes **MoCS** various command-line arguments.

Argument	Description
`cp`	The SDN controller program that is input to the model, as discussed in Section 2.1 of the paper
`sr`	Whether bitwise state representation is used or not
`por`	Whether partial-order reductions are being applied or not
`switches`	Number of switches in the topology
`hosts`	Number of hosts in the topology

Note that the actual values of these parameters are subject to limitations as discussed in the section below. Limitations on the number of switches and hosts come from the fact that their respective underlying network topologies are currently hardcoded. The output returned by the `mocs` command includes (1) an affirmative response if the property holds, otherwise negative (2) the number of the states explored, (3) the CPU user time spent, (4) the memory used, and (5) the throughput. For example, below is the output for a 1-switch, 2-hosts topology with the stateful firewall controller program, having both optimisations, `sr` and `por`, turned on.

```
mocs@mocs-VirtualBox:~$ mocs fw wB wP 1 2

 -- Formula is satisfied.
 -- States explored      : 592 states
 -- CPU user time used   : 120 ms
 -- Resident memory used : 8508 KiB
 -- Throughput           : 4933 states/s

mocs@mocs-VirtualBox:~$
```

Reproducing the Experimental Evaluation

States explored, verification throughput and memory footprint

The values of the arguments used for the experimental evaluation reported in Figures 4-6, are bound to the ones shown in the table below.

	cp	sr	por	switches	hosts
mocs	fw	{wB | woB | 0}	{wP | woP | 0}	{1..4}	2
mocs	ml	{wB | woB | 0}	{wP | woP | 0}	{3..10}	{2..6}
mocs	ssh	{wB | woB | 0}	{wP | woP | 0}	{2..10}	2

where `ml` stands for MAC-learning switch, `fw` for stateful firewall and `ssh` for stateless firewall, the three different SDN controller we experimented with in the paper. The curly braces indicate that the user must choose one and only one of the items inside the braces. The notation `mocs ml {wB | woB | 0} {wP | woP | 0} {3..10} {2..6}`, for example, says that the command `mocs` must be followed by either `wB`, `woB` or `0` for state representation, by either `wP`, `woP` or `0` for partial-order reduction, the number of switches (currently between 3 and 10), and the number of hosts (currently between 2 to 6). When the `0` values for state representation (`sr`) and partial-order reduction (`por`) are used, the resulted model is that of the optimised version of Kuai (https://github.com/t-saideep/kuai).
Each combination of the arguments `cp`, `sr`, `por`, `switches` and `hosts` from the table above, points to one `xml` file in a leaf-subfolder within `~/inputs/Scaling_up` in the submitted artifact.

Immediate-subfolder naming convention: Each immediate subfolder in:
`~/inputs/Scaling_up`, contains data related to a specific controller program as named, i.e., either `ssh`, `ml`, `fw`.

Leaf-subfolder naming convention: The end-subfolders are named conventionally in order to provide a preview of the content. They are organised by (1) controller program name (`ssh`, `ml`, `fw`), and (2) whether the optimisations are on/off: `wB/woB/0`, `wP/woP/0`.

File naming convention: The `xml` input files are also named conventionally and they are organised by (1) controller program name (same format as their immediate parent directory), and (2) topology setup (data plane instances), where *mn* denotes a network of *m* switches and *n* hosts as shown in the `~/inputs/Dataplane_topologies` files.

Examples

`mocs ssh wb wp 3 2` will invoke the file:
`~/inputs/Scaling_up/SSH/SSH_wBwP/SSH32.xml` which refers to an optimised model (both efficient state representation and partial-order reductions are turned on) in a 3-switches, 2-hosts forwarding network controlled by a stateless firewall.
`mocs fw wB wP 1 2` will invoke `~/inputs/Scaling_up/FW/FW_wBwP/FW12.xml`, a stateful firewall with 1 switch and 2 hosts, and optimised semantics.
`mocs ml 0 0 3 2` will invoke `~/inputs/Scaling_up/ML/ML_Kuai/ML32.xml` which is a Kuai's model (with all the Kuai's optimisations) in a 3 x 2 topological setting with MAC Learning Switch controller program.

Model Expressivity

For reproducing the paper results regarding the model expressivity, we use `mocs_opt` and `kuai`, which are instances of `mocs` for specific configurations. `mocs_opt <cp>` is an instance of `mocs` for the fully optimised version of **MoCS** models (i.e., both state representation and partial order reduction optimisations are turned on). `kuai <cp>` is an instance of `mocs` for Kuai-models (with Kuai's optimisations turned on). Both commands take as an argument the controller program and embed 2-hosts, 2-switches topologies, apart from the "nesting_level" example which runs on a 3 x 3 topology.
In the paper, we looked at three different controller programs (described in Appendix B in our paper), along with respective properties to be checked. In the first two examples, the buggy versions of said controllers contained a bug that could not be discovered by Kuai due to its expressivity limitations. **MoCS** could discover the bug in both examples. The list of commands for reproducing these findings (i.e. discover the bugs in the buggy versions and verify the correctness of the SDN with respect to the given property) is shown below. The output of `mocs_opt` and `kuai`, resp., is identical to `mocs`.

```
mocs_opt   CM_ordering_buggy          // will discover bug (desired outcome)
mocs_opt   CM_ordering_correct        // property will be verified (desired outcome)

mocs_opt   wrong_nesting_level        // will discover bug (desired outcome)
mocs_opt   correct_nesting_level      // property will be verified (desired outcome)

kuai       CM_ordering_buggy          /* property will be verified (bug not
                                       * discovered - wrong outcome)
                                       */

kuai       wrong_nesting_level        /* property will be verified (bug not
                                       * discovered - wrong outcome)
                                       */
```

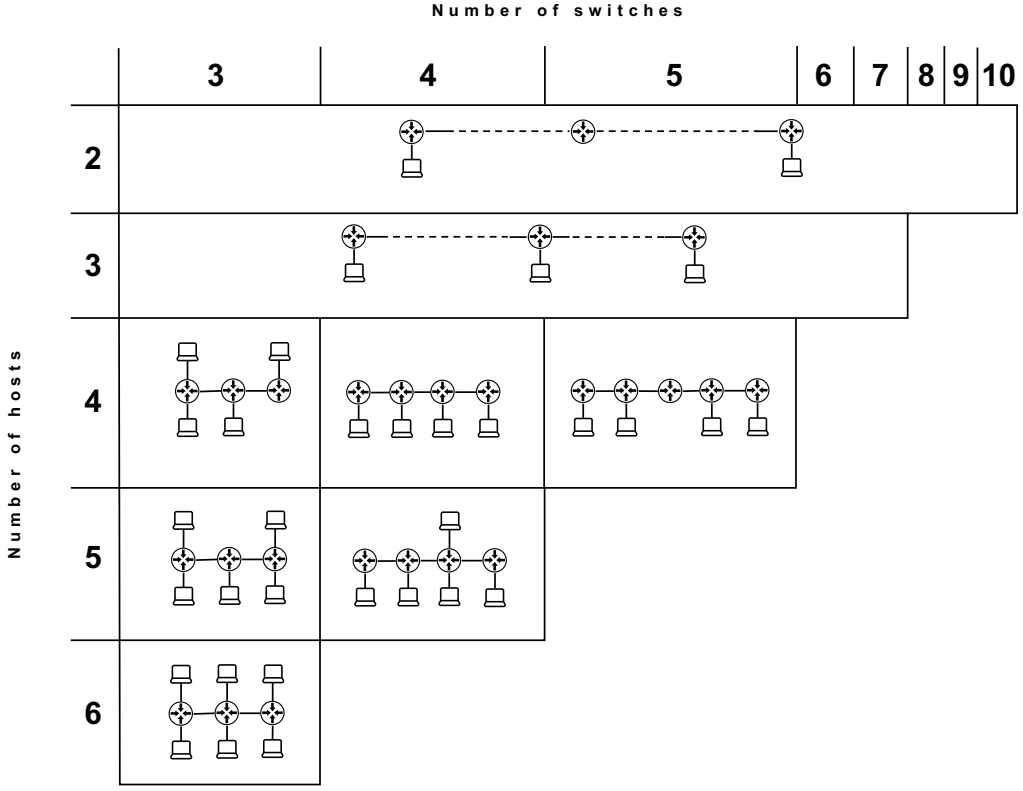
In the third example (*(In)consistent updates*), the bug can be due to either (1) the inability of the model to express barrier-response messages (i.e. as in Kuai), or (2) the omission of their usage when barrier-response messages are supported (i.e. as in **MoCS**). The bug can only be fixed by updating the controller program to change its state according to the incoming barrier-response messages, which, obviously, cannot be done with Kuai. The findings in the paper can be reproduced with the following commands.

```
mocs_opt   inconsistent_updates       // will discover bug (desired outcome)
mocs_opt   consistent_updates         // property will be verified (desired outcome)
```

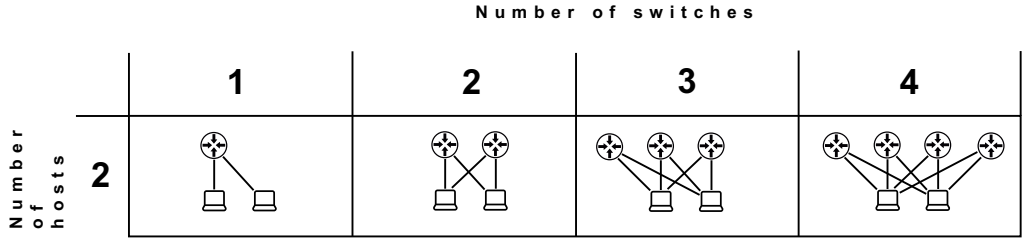
Note that all the above executions are very quick and do not require any significant memory resources.

Dataplane topologies

All the network setups (data plane topological instances) used to evaluate **MoCS** for (1) the MAC learning and stateless firewall applications, and (2) the stateful firewall are depicted in the below two figures, (*) and (**), respectively.



(*) Network topologies for verifying absence of loops in the MAC address learning application. The topology setups for the stateless firewall follow the pattern of those with two switches.



(**) Network topologies for the stateful firewall.