Formal modelling of software defined networking

Formal modelling of software defined networking

Vashti Galpin

Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh
Vashti.Galpin@ed.ac.uk, ORCID: 0000-0001-8914-1122

Abstract. This paper investigates the application of CARMA, a recently developed quantitative process-algebra-based modelling language, to the stochastic modelling of software defined networking (SDN). In SDN, a single controller (or hierarchy of controllers) determines the behaviour of the switches that forward traffic through the network, and it is used in a variety of settings including cloud and data centres. This research is the initial phase of developing a methodology for agile formal modelling of performance and security aspects of SDN, and focusses on the fat-tree network topology. The results demonstrate that the CARMA language and its software tools which include the MultiVeStA statistical model checker provide a good basis for modelling SDN.

1 Introduction

Traditionally, network modelling has been done by emulation on virtual machines using a tool called mininet [29] or simulation using a tool such as ns-3 [9, 26]. Both of these approaches consider the full network stack, which can be very expensive in terms of initial setup, as well as computational resources to execute. Other simulation approaches allow for some abstraction [6, 8, 30] from these details, in order to reduce these overheads and various formal approaches have also been suggested [3, 4, 7, 27, 33–36, 40, 42], most of which have no quantitative features.

Our goal is to provide a novel approach to modelling the behaviour of networks at a moderately high level of abstraction but with the ability to measure performance, something that is missing from most formal approaches for networks. This will still allow for a quantitative assessment of network behaviour which is crucial to evaluate different configurations but provide a lighter-weight approach that the full-stack emulation and simulation methods. Our approach models individual packets traversing a network but abstracts from lower level concerns of the network stack. This may reduce what questions can be answered by the model; however, it will still allow many questions of interest such as packet latency, to be answered much more quickly than the traditional full stack emulation and simulation approaches, and hence provides an alternative approach. As is well known, formal modelling of computer systems has multiple benefits including the ability to reason about a system before it is built, and to conduct experiments using a model of an existing system without disrupting the system itself.

This paper considers how an existing probabilistic modelling language can be used to simulate the behaviour of software defined networking (SDN) [15] at the packet level, allowing for the investigation of performance and security properties, as well as

trade-offs between these properties. Specifically, we work with the quantitative formal modelling language CARMA and examine how CARMA and CaSL, the textual language of the CARMA Eclipse Plug-in tool [25, 32] support this type of modelling. MultiVeStA [38] is integrated into a command-line version of the tool, allowing statistical model checking of CARMA models on top of simulation.

The textual language of the CARMA tool provides an explicit syntax and a location type for expressing location with respect to a structure that describes discrete space. This motivated the choice of CARMA to model physical network topology and allows for a parametric approach to network topology description. Also importantly for practical application of CARMA, the tool also provides a rich choice of attribute types, including integer, real, enumerated types, Boolean, and finite lists and sets of these types. Functions can be defined over all data types, to support programmatic aspects of models, and goes beyond process-algebra-style behaviour and interaction.

Our experiments show that there is a good match between the discrete space syntax provided by CARMA and modelling network topology. This makes it possible to separate network topology (and traffic) from the definition of generic network elements such as hosts, switches and controllers. This has advantages in terms of speed of model construction as well as ease of debugging models. The three major contributions of this research are as follows. First, it provides an assessment of CARMA for modelling network performance and security in SDN through the development of a model that permits packet-level modelling. The model contains generic controller, switch and host components that allow for the controller to send flow table rules to switches which are then able to direct packets from one host to another through the network. Packets are modelled explicitly and their header content is used by switches to determine how they should be handled depending on the flow rule that applies. Furthermore, the model is parametric with respect to the network description, allowing fast development of models with different topologies.

Second, it allows for experimentation with the fat-tree topology in a SDN setting that considers the scalability of the topology with respect to packet latency which is a standard measure of network performance, considering both uniform and MapReduce traffic; with the goal of determining the packet rates at which the network become congested and can no longer operate at line speed (the speed of the underlying network connections) because of queues at switches. The performance cost of mitigation of attacks is also considered. Finally, it explores the use of MultiVeStA for statistical model checking of switch queue sizes, allowing for the exploration of the parameter space of MapReduce traffic patterns, thereby integrating the use of different formal methods approaches.

2 Background

This research contributes the first phase of the development of a general methodology for modelling of networks and network security, and it is necessary to select a particular case study for exploration of the potential of the approach. We have chosen to consider the use of software defined networking (SDN) in data centres. This specifies the focus and helps identify suitable examples with which to work. SDN is an ideal setting since it

is an industry standard whose deployment is wide-spread and increasing. As networking requirements become more complex in cloud and data centre scenarios, SDN provides a different approach based on a full network overview compared to other approaches. One of the complexities that must be addressed is security and SDN offers opportunities and challenges in this domain [12]. Furthermore, there is an ongoing need for assessment of SDN performance due to the range of implementations and switch types [17].

The distinct roles of network elements in SDN maps well to CARMA components which describe behaviour with the addition of store, allowing for internal state, which is not often a feature of process algebras. Focussing on data centres allows the consideration of regular topologies which is a good starting point for modelling. Regular topologies also make large networking scenarios possible programmatically and we will show later in the paper how CARMA supports this.

2.1 Software defined networking

In traditional networking, routers direct packets and have enough knowledge about the state of the network to make forwarding decisions. Software defined networking [15] takes a very different approach whereby the network switches are provided with flow rules (by the controller which has an overview of all network behaviour) that specify how packets should be directed. Each switch has a flow table that is stored in fast (but expensive) ternary content-addressable memory (TCAM) which allows for fast look-up. When a packet arrives at a switch, its header is compared with the flow table entries. These entries may contain wildcards, and different packet headers may match a single flow rule. If a match is found, the action specified by the rule is followed and counters for the rule are updated, and if not the packet header is sent to the controller (which may add a new rule to the switch for that packet header). The two most common actions for a rule are **forward** on a port number, or **drop** where the packet is not forwarded.

Rules in the flow table can be divided into proactive and reactive. Reactive rules are those that are installed when the controller must decide what to do with a packet that does not match the rules at a switch. By contrast, proactive rules are those that are installed by the controller as a switch becomes active, based on an overview of the network topology and specific choice of a single route between each pair of hosts. We focus on proactive rules and the performance of a balanced routing over a network. This is not a limitation of the modelling as a variant of this model has been used to consider reactive rules in the evaluation of an attack mitigation [10].

The controller makes decisions programmatically about the flows through the network, determined routes from information about network topology, existing traffic and updating routes when necessary. This route choice can have different aims such as performance (efficient use of network bandwidth) or security (for example, mitigation for covert channel attacks [41]).

2.2 Fat-tree topology

Our focus is on the fat-tree topology [2] that is used in data centres and is suitable for use with SDN. In a standard tree topology, there may be a single switch at the top level of the tree (the core) and this is a bottleneck. By contrast, the fat-tree topology is

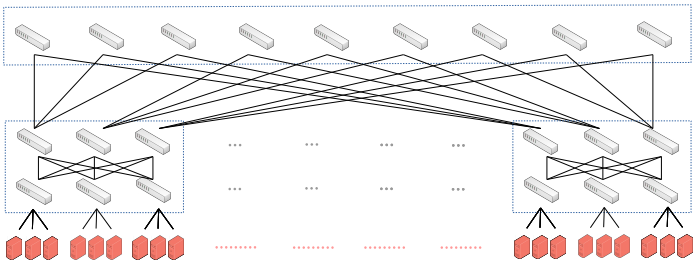


Fig. 1. Fat tree with 6-port switches

based on k-port switches and provides more than one core switch. Figure 1 illustrates the fat-tree topology obtained using 6-port switches The top block of nine switches are the *core*. There are then k pods, each containing k switches. The layer of switches immediately under the core is referred to as the *aggregate* and the layer before the hosts, the *edge*. The 6-port topology supports 54 hosts and there are 9 routes between each pair of hosts that are in different pods, as can be seen from the figure by considering the first host of the first pod and the first host of the last pod. The controller is not included in this diagram and there is research into where controllers should be placed for best performance [22]. In this research, we abstract from these details, as discussed later.

More generally, a 3-level fat tree based on commodity switches with k ports has a core of $k^2/4$ switches and k^2 switches at the aggregate and edge level. This allows for the support of $k^3/4$ hosts. Between each pair of hosts, there are $k^2/4$ routes, one through each switch of the core. The use of commodity switches allows for a cheap but efficient topology, and it is well suited for SDN because of multiple routes.

Over and above modelling topologies, we need to model traffic patterns as well. In the experiments, we will consider two traffic patterns: one where there is traffic between all hosts where we consider at what traffic levels, the packet latency becomes too high. The second considers the MapReduce pattern where many hosts communicate with a single host to convey the results of calculations done in parallel.

3 Related work

In the case of SDN, mininet [29] allows for emulation by modelling actual network behaviour on multiple virtual machines. A limitation of mininet is that it runs in real-time as an emulator and hence does not scale to large systems. Network simulators such as ns-3 [9, 26] simulate the full network stack behaviour. Both of these approaches are costly in terms of initial set-up and have steep learning curves. An alternative is a much more abstract approach such as CloudSim which is used to model large SDN data centres [6] but this is an understanding of network performance before building the model, and hence is not suitable for the type of network performance and security modelling proposed here. A different approach is taken using TopoGen in modelling of SDN topologies [30] where the focus of the tool is in supporting large topologies

and then using hybrid modeling, simulation and control of data networks based on a hybrid DEVS (Discrete Event System Specification) formalism where some packets are modelled explicitly and some as flows [8].

Various formal approaches have been suggested such as NetKat [4], Veriflow [27] and others [3, 7, 33–36, 40, 42]. Probabilistic NetKat [39] is the closest to our approach but is limited to a time-homogeneous approach. Using CARMA, it is possible to consider behaviour over time as parameters vary, allowing for the dynamic modelling of the effects together with mitigation of attacks, as illustrated later. Some process algebras have also been proposed for network modelling such as [28] but most focus on wireless or ad hoc networks and are not quantitative such as [16]. In the case of the spatial extension of PEPA (which is a stochastic process algebra that influenced the development of CARMA) [18], CARMA offers a much richer way to specify behaviour.

4 Overview of CARMA

CARMA (Collective Adaptive Resource-sharing Markovian Agents) is a process-algebra-based quantitative modelling language developed for the modelling of collective adaptive systems with explicit support for the modelling of discrete space, as well as separate specification of an environment. [25, 32]. It has roots in process algebras developed for performance evaluation such as PEPA [23] and biological modeling such as Bio-PEPA [11], and has (time-inhomogeneous) continuous-time Markov chain semantics. It has richer interaction than PEPA or Bio-PEPA, and uses attribute-based communication similar to that of SCEL [14] and AbC [1]. It has been used to model taxi movement [24], carpooling [43], ambulance deployment [19] and pedestrian mobility [21].

The language has two forms: the mathematical definition of CARMA [25] and the language CaSL [20] of the CARMA Eclipse Plug-in (available at quanticol.sourceforge.net). Both will be used in this paper. The former will be used to provide abstract presentations of the models, and the latter to describe how measures and spatial concepts are defined with the software tools, and thereby illustrate the power of the implementation with respect to network topology modelling.

The basic behavioural unit of a CARMA model is a *component* which consists of communicating behaviour specified using process-algebra-style prefixes (actions), an initial behaviour and a store of attributes that characterise the component. Interaction between components can be unicast or broadcast. The components of a model form a *collective* which then operates within an *environment*. The environment includes a global store, and updates to elements in the store are triggered by actions performed by components. It also specifies the rates and probabilities at which actions are performed, and allows for new components to be added to the collective when given actions are performed by components. In the context of the SDN model, packet rates will appear in the environment as will global attributes to calculate packet latency.

To understand the basic behaviour of component, consider a component that has three attributes v, x and y in its store. Behaviour in this component can be specified in the following form. Here $\top$ indicates true and $\bot$ false.

$$A \stackrel{\text{def}}{=} \mathsf{signal}[\top]\langle v\rangle\{a \leftarrow a-1\}.A + \\ \mathsf{new\_signal\_count}^{\star}[\mathsf{my}.b < b](new\_a)\{a \leftarrow new\_a\}.A + \\ [a=0]\mathsf{finished}^{\star}[\bot]\langle\rangle.\mathbf{nil}$$

Process A can repeatedly send out a signal of v to one other component (and wait until it is received) at which point the value of a is decreased by one. It can also receive a broadcast communication (indicated by the asterisk) from any other component which has a larger b value than it, which communicates a value that is then stored as a. This is attribute-based communication: A can only "hear" from components with a larger b value. However, if a becomes 0, then the process can perform an internal action and become the process with no behaviour. The use of a broadcast action with a false predicate leads to an internal action since no other component can satisfy the predicate, and broadcast is not blocking. Thus the action happens without interaction. As will be seen in the SDN model, sometimes an attribute is updated by calling a function, and no interaction with other components is necessary, and hence we use this form of action.

The component containing behaviour A, say *CompA*, can be described as a component that attempts to communicate (by unicast) the value v a certain number of times after which it ceases communication. However, imagine that there are other components (not specified here) which are allowed to communicate a new value of a to *CompA*. This can either reduce or increase how many more times the value v is sent by *CompA*. The other components which can communicate new values must have larger b values (which could be a priority) than *CompA* to successfully change a.

Formally, components have the syntax $C ::= \mathbf{0} \mid (P, \gamma)$ where $\mathbf{0}$ is the null component, P is a process that describes behaviour and γ is the store. Stores map from *attribute names* to *basic values*. The syntax of processes that define the behaviour of components are specified by the following

$$\begin{array}{rl|rl} P,Q ::= & \mathbf{nil} \mid \mathbf{kill} & act ::= & \alpha^{\star}[\pi]\langle\overrightarrow{e}\rangle\sigma \mid \alpha\ [\pi]\langle\overrightarrow{e}\rangle\sigma \\ \mid & act.P \mid P+Q & \mid & \alpha^{\star}[\pi](\overrightarrow{x})\sigma \mid \alpha\ [\pi](\overrightarrow{x})\sigma \\ \mid & P \mid Q \mid [\pi]P & e ::= & a \mid \mathsf{my}.a \mid x \mid v \mid \mathsf{now} \mid \cdots \\ \mid & A \qquad (A \stackrel{\text{def}}{=} P) & \pi ::= & \top \mid \bot \mid e_1 \bowtie e_2 \mid \neg\pi \mid \pi \wedge \pi \mid \cdots \end{array}$$

where

- α is an *action type*; π is a *predicate*;
- e is an *expression*; x is a *variable*; $\overrightarrow{\cdot}$ indicates a sequence of elements;
- a is an *attribute name*; v is a *basic value*;
- σ is an *update* defined by a function from Γ to $Dist(\Gamma)$ where $Dist(\Gamma)$ is the set of probability distributions over Γ. This allows for stochastic updates.

The behaviour includes the absence of behaviour **nil**, the ability to remove a component from the collective **kill**, action prefix $act.P$, choice $P+Q$, parallel composition $P \mid Q$, predicate prefix $[\pi]P$ where the behaviour as P is only available if the guard π defined over the component's attributes is true, and constant definition. Expressions include now for the current simulation time and my.a which refers to the value of the attribute a in the current component. When referring to an attribute shared by a two components, my.a allows for distinction between the two in predicates that constrain interaction.

The different prefixes specify the type of interaction.

Broadcast output: $\alpha^\star[\pi]\langle\overrightarrow{e}\rangle\sigma$
Broadcast input: $\alpha^\star[\pi](\overrightarrow{x})\sigma$
Unicast output: $\alpha[\pi]\langle\overrightarrow{e}\rangle\sigma$
Unicast input: $\alpha[\pi](\overrightarrow{x})\sigma$

Here α is an action name, π is a predicate over attributes of the sender and the receiver, and σ specifies attribute updates. For output, $\overrightarrow{e}$ is a list of output expressions, and for input, $\overrightarrow{x}$ is a list of variables, as is standard. Broadcast actions are indicated by the presence of an asterisk. As mentioned above, an internal action has the form $\alpha^\star[\bot]\langle\rangle\sigma$.

The predicates after the action name in a prefix determine who takes part in the communication. Rates, probabilities and weights associated with an action name are recorded in the environment element of the model and may depend on attributes of the sender (in the case of broadcast which is non-blocking) and on the sender and receiver (in the case of unicast which is blocking). We use predicates in the SDN model to ensure unicast communication only occurs between components that are directly connected in the network and this is specified by the space description.

A collective N consist of either a component C or the parallel composition of two collectives, $N ::= C \mid N \parallel N$, and a CARMA model then consists of a collective together with an environment $S ::= N$ **in** $\mathscr{E}$. The environment collects together all of the information necessary for the collectives to operate including rules that regulate the system such as rates of interaction and probabilities that interaction may occur, as well as global information. The environment consists of two elements: a *global store* γ_g to record the value of global attributes, and an *evolution rule* ρ. This is a function which, depending on the *current time* (using now), the global store and the current state of the collective returns four functions defined on stores and action names. These are known as the *evaluation context*.

Probabilities: $\mu_p(\gamma_s, \gamma_r, \alpha)$ determines the probability that a component with store γ_r can receive a message from a component with store γ_s when α is executed;

Weights: $\mu_w(\gamma_s, \gamma_r, \alpha)$ determines the weight allocated to α executed by a component with store γ_r receiving a message from a component with store γ_s. This weighting determines the probabilities between different unicast actions.

Rates: $\mu_r(\gamma, \alpha)$ provides the execution rate of action α executed at a component with store γ;

Updates: $\mu_u(\gamma, \alpha)$ determines the updates on the environment (global store and collective) induced by the execution of action α at a component with store γ. The execution of an action can modify the values of global variables and also add new components to the collective.

Figure 6 provides an example of this evaluation context as used in the SDN model. For the rates and updates, the function is expressed as a series of cases based on the particular action involved. For the SDN model, there is little use of the component stores in the definitions. The only explicit occurrence is in the update for for the action log_packet*, although there are more occurrences in the elided details of the rates for packet generation and the traffic patterns which could be determined by the identity of the source or destination of a packet.

The operational semantics of CARMA specifications are defined through transition relations. The semantic rules can be found in [32]. These relations are defined in the FUTS style [13] and are described using a triple $(N, \ell, \mathscr{N})$ where the first element is a component, or a collective, or a system; the second element is a transition label; and the third element is a function associating each component, collective, or system with a non-negative number. If this value is positive, it represents the rate of the exponential distribution characterising the time needed for the execution of the action represented by ℓ. A zero value is associated with unreachable terms. FUTS style semantics are used because it makes explicit an underlying (time-inhomogeneous) Action Labelled Markov Chain, which can be simulated with standard kinetic Monte Carlo algorithms.

Two further elements are important for modelling with CARMA. Measures define the outputs of a CARMA model when it is simulated. A space description defines the discrete space model over which a CARMA model will operate. It defines a weighted graph structure, and each component can be located at a node in this graph. To aid clarity of presentation, these two elements will be presented using CaSL, the textual language of the software tools, in the next section where the model is introduced.

5 The SDN model

CARMA is a rich formalism developed for a specific purpose; however it is applicable to a variety of systems. For SDN, an important goal of the model is to be parametric with respect to the network specification. Therefore, generic components model various aspects of the system and there is no specification of the network details outside of the portion of the model that describes the network topology and traffic parameters. Thus we need only define four generic components for our model: host, switch, controller and timer. For some scenarios, deterministic time delays are required and the timer component supports their modelling. In the model presented here, it determines when data should be collected from the switches by the controller. Each switch has a unique location and each host is located at a switch.

The interaction between the components is illustrated in Figure 2 where components with a single border have a single occurrence, and with double borders, multiple occurrences. The CARMA components for all four are given in Figures 3 and 4. The notation x_i refers to element i of array x, except in the case of action names, where action_i refers to indexed action names. The model presented is an abstraction of the CaSL model developed (for reasons of space) and some aspects are only mentioned in passing. Furthermore, this model concentrates on proactive flow rules, and the procedure for dealing with packets that do not match a rule are not described.

The controller and switch components call various functions to support their behaviour. Examples of controller functions are *SelectRoutes* which generates specific routings from the network topology, *CalcStats* and *CalcFlow* which takes switch counter information and calculates overall flows between hosts which can be then used by functions to update the routing array. Additional functionality within the SDN paradigm can be added by providing new functions rather than modifying the components, in most cases. It is also possible to add novel behaviour that SDN does not currently support such as probabilistic choice of rules to decide forwarding of packets in switches. With

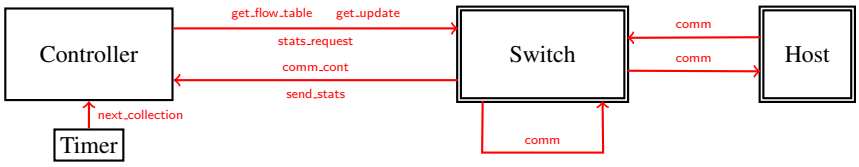


Fig. 2. Interaction of generic components

just these four component types and a separate network description, it is possible to create large network examples for analysis.

The host component (Figure 3) allows for different traffic patterns specified as separate parallel components. An example of this is MapReduce which will be used in the experiments. Details of these have been elided for reasons of space. However, the basic idea is that certain traffic patterns will either be switched on or off as indicated by a Boolean attribute in the component. When a pattern is on, a packet corresponding to that traffic pattern can be sent. The host component repeatedly generates and sends packets, keeping count of the number. For measurement of packet latency, the time a packet is sent is included in the packet itself, together with the source, destination and protocol. In parallel, the host receives packets for which it is the destination and counts them. Both of these are done using the comm action together which a predicate that ensures that communication can only occur between connected network elements. This is discussed further in Section 5.2 below.

The switch component (Figure 3) first receives its rule table from the controller and then processes incoming packets that are put into a queue when they arrive. For each packet, a matching rule is sought from the flow table where the flow table is an array of rule records which include source host identifier, destination host identifier, protocol, action to be done and counter to record how many times the rule is used. The action associated with the rule is applied: a packet can be dropped, sent to the controller, or forwarded. Parallel components wait for communication from the controller for statistics requests and rule updates, and set appropriate flags. Packet processing must be paused to update rules or send the flow table to the controller whenever an update is available or a request has been sent. Switch components use a single action comm to communicate with each other and hosts, and the significance of this will be discussed in Section 5.2.

The controller component uses the function *GenTop* to generate all possible routes between each pair of switches from the space description (see Section 5.1 for details of this description). From this collection of routes, a routing array is obtained using the function *SelectRoutes* to describe the route that a packet from one host to another will take. In SDN, exactly one route is chosen for a flow of packets from one host to another. In the model, the routes chosen can be balanced with respect to the number of hosts, so flows are evenly spread across core switches. They can also be unbalanced where a single core switch is used for all flows, or flows can be randomly assigned to different routes. The function *ConstructTables* is used to construct the flow tables for each switch which are then communicated to each switch, one by one. The controller component then interacts with the timer component to wait until it is time for the next

Store of *Host* component:

$host$	host identifier
loc	location of switch that host is attached to
in	number of packets input
out	number of packets output
$\ldots$	traffic pattern information for each traffic type

Behaviour of *Host* component:

$$Output \stackrel{\text{def}}{=} \sum_i \mathsf{gen\_pkt\_type}_i^{\star}[\bot]\langle\rangle\{out\_pkt \leftarrow GenPktType_i(host, \mathsf{now}, \ldots)\}.Send$$

$$Send \stackrel{\text{def}}{=} \mathsf{comm}[ReceiverIsConn(\mathsf{my}.loc, loc)]\langle out\_pkt\rangle\{out \leftarrow out + 1\}.Output$$

$$Input \stackrel{\text{def}}{=} \mathsf{comm}[SenderIsConn(\mathsf{my}.loc, loc) \wedge Dest(p) = host](p)\{in\_pkt \leftarrow p\}.Log$$

$$Log \stackrel{\text{def}}{=} \mathsf{log\_packet}^{\star}[\bot]\langle\rangle\{in \leftarrow in + 1\}.Input$$

$$TP_i \stackrel{\text{def}}{=} \text{traffic pattern behaviour for packet type}_i$$

Initial state of *Host* component: $Output \mid Input \mid TP_1 \mid \ldots \mid TP_n$

Store of *Switch* component:

loc	switch identifier
$flow\_table$	table of flow rules
q	queue of input packets
upd	if true then rule updates are available for installation
$stats$	if true then a stats request is pending

Behaviour of *Switch* component:

$$Start \stackrel{\text{def}}{=} \mathsf{get\_flow\_table}[SenderIsContr(loc)]\langle t\rangle\{flow\_table \leftarrow t\}.Process$$

$$Process \stackrel{\text{def}}{=} [size(q) > 0]\mathsf{get\_head}^{\star}[\bot]\langle\rangle\{pkt \leftarrow head(q), q \leftarrow tail(q)\}.Match$$

$$Match \stackrel{\text{def}}{=} \mathsf{find\_rule}^{\star}[\bot]\langle\rangle\{rule \leftarrow FindRule(flow\_table, pkt)\}.Count$$

$$Count \stackrel{\text{def}}{=} \mathsf{incr\_count}^{\star}[\bot]\langle\rangle\{flow\_table \leftarrow IncrCount(flow\_table, rule)\}.Act$$

$$\begin{aligned} Act \stackrel{\text{def}}{=}\ & [Action(rule) = \text{DROP}]\mathsf{skip}^{\star}[\bot]\langle\rangle.Update + \\ & [Action(rule) = \text{CONT}]\mathsf{comm\_cont}[ReceiverIsContr(loc)]\langle pkt\rangle.Update + \\ & [Action(rule) = \text{FORW}]\mathsf{comm}[ReceiverIsConn(\mathsf{my}.loc, loc)]\langle pkt\rangle.Update \end{aligned}$$

$$\begin{aligned} Update \stackrel{\text{def}}{=}\ & [upd]\mathsf{rule\_update}^{\star}[\bot]\langle\rangle\{flow\_table \leftarrow RuleUpdate(rt, u), upd \rightarrow \bot\}.Stats + \\ & [\neg upd]\mathsf{skip}^{\star}[\bot]\langle\rangle.Stats \end{aligned}$$

$$\begin{aligned} Stats \stackrel{\text{def}}{=}\ & [stats]\mathsf{send\_stats}^{\star}[ReceiverIsContr(loc)]\langle flow\_table\rangle\{stats \rightarrow \bot\}.Process + \\ & [\neg stats]\mathsf{skip}^{\star}[\bot]\langle\rangle.Process \end{aligned}$$

$$Listen \stackrel{\text{def}}{=} \mathsf{comm}[SenderIsConn(\mathsf{my}.loc, loc)](p)\{q \leftarrow Append(q, p)\}.Listen$$

$$StatsReq \stackrel{\text{def}}{=} \mathsf{stats\_request}[SenderIsContr(loc)]()\{stats \leftarrow \top\}.StatsReq$$

$$UpdRec \stackrel{\text{def}}{=} \mathsf{get\_update}[SenderIsContr(loc)](u)\{upd \leftarrow \top, flow\_table \leftarrow u\}.UpdRec$$

Initial state of *Switch* component: $Start \mid Listen \mid StatsReq \mid UpdRec$

Behaviour of *Timer* component:

$$NextCollect \stackrel{\text{def}}{=} \mathsf{next\_collection}[loc = [Con]]\langle \mathsf{now}\rangle.NextCollect$$

Initial state of *Timer* component: $NextCollect$

Fig. 3. Host, Switch and Timer components

Store of *Controller* **component:**

loc	controller identifier, always *Con*
top	topology, describes all routes between pairs of switches
$routing$	routing array, describes a single route between any pair of hosts
$tables$	array of flow tables, one entry for each switch
$last\_t$	time of last statistics collection
sw_data	array of switch data, one entry for each switch
nwf_flow	total flow between each pair of hosts
sw_upd	array of switch updates, one for each switch

Behaviour of *Controller* **component:**

$$
\begin{aligned}
\mathit{StartCon} &\stackrel{\text{def}}{=} \mathsf{gen\_top}^{\star}[\bot]\langle\rangle\{\mathit{top} \leftarrow \mathit{GenTop}()\}.\mathit{Routing}\\
\mathit{Routing} &\stackrel{\text{def}}{=} \mathsf{choose\_routes}^{\star}[\bot]\langle\rangle.\{\mathit{routing} \leftarrow \mathit{SelectRoutes}(\top)\}.\mathit{SwInit}\\
\mathit{SwInit} &\stackrel{\text{def}}{=} \mathsf{construct\_tables}^{\star}[\bot]\langle\rangle\{\mathit{tables} \leftarrow \mathit{ConstructTables}(\mathit{routing}), i \leftarrow 0\}.\mathit{SwInst}\\
\mathit{SwInst} &\stackrel{\text{def}}{=} [i < S]\mathsf{get\_flow\_table}[loc = [i]]\langle \mathit{tables}_i\rangle\{i \leftarrow i+1\}.\mathit{SwInst} +\\
&\quad\ [i = S]\mathsf{skip}^{\star}[\bot]\langle\{i \leftarrow 0\}\rangle.\mathit{Wait}\\
\mathit{Wait} &\stackrel{\text{def}}{=} \mathsf{next\_collection}[t > \mathit{last\_t} + \mathit{collect\_interval}](t)\{\mathit{last\_t} \leftarrow t\}.\mathit{Request}\\
\mathit{Request} &\stackrel{\text{def}}{=} [i < S]\mathsf{stats\_request}[loc = [i]]\langle\rangle.\mathit{Collect} +\\
&\quad\ [i = S]\mathsf{calc\_flow}^{\star}[\bot]\langle\rangle\{\mathit{nwf\_flow} = \mathit{CalcFlow}(\mathit{nwf\_flow}, \mathit{sw\_data}), i \leftarrow 0\}.\mathit{Analysis}\\
\mathit{Collect} &\stackrel{\text{def}}{=} \mathsf{send\_stats}^{\star}[loc = [i]](d)\\
&\qquad \{\mathit{sw\_data}_i \leftarrow \mathit{CalcStats}(d, \mathsf{now}, \mathit{sw\_data}_i), i \leftarrow i+1\}.\mathit{Request}\\
\mathit{Analysis} &\stackrel{\text{def}}{=} (\ldots \text{decisions about changes to routing array based on } \mathit{nwf\_flow} \text{ resulting in}\\
&\quad\ \text{new rules for switches stored in } \mathit{sw\_upd}\ldots).\mathit{SwUpd}\\
\mathit{SwUpd} &\stackrel{\text{def}}{=} [i < S]\mathsf{rule\_update}[loc = [i]]\langle \mathit{sw\_upd}_i\rangle.\mathit{SwUpd} +\\
&\quad\ [i = S]\mathsf{skip}^{\star}[\bot]\langle\rangle\{i \leftarrow 0\}.\mathit{Wait}
\end{aligned}
$$

Initial state of *Controller* **component:** *StartUpCon*

Fig. 4. Controller component

collection of statistics, which is also done one-by-one from each switch. The counts in the flow tables of each switch are compared with the counts at the previous collection and traffic flows between hosts are obtained which can then be used to determine switch updates. As mentioned previously, this model considers proactive aspects of SDN rather than reactive, and hence the details of switch updates have not been included.

5.1 Space syntax for network description

To specify a network description in a CARMA SDN model requires the expression of the network topology in the space syntax which is independent of the generic components defined above, as well as traffic information for the network which is captured in two arrays for a specific traffic pattern. The first specifies the rate at which each host generates packets, and the second specifies the distribution of destinations for the packets, allowing for stochastic behaviour to be defined.

There is also second level of parametericity in CARMA SDN modelling. The size and shape of various topologies can be parametric and the space description can be defined to take this into account. For example, in the fat-tree topology, the parameter that

```
const k = 6; // number of ports in each switch
space kPort_FatTree_Pod (){
  universe <int x>
  nodes { for i from 0 to k {[i];} }
  connections { for i from k/2 to k {
      for j from 0 to k/2 {
        [i] -> [j] { port=j };
        [j] -> [i] { port=i }; } } }
}
const Host_Switch = [: [3],[3],[3],[4],[4],[4],[5],[5],[5] :];
const Host_Port = [: 3,4,5,3,4,5,3,4,5 :];
```

Fig. 5. Space specification for one pod of Figure 1 (partially parameterised)

specifies the size of the network is the number of ports in the type of switch used. For the experiments described later, the network topology description is parametric in this number, and increasing the parameter, increases the size of the network without time-consuming model updates as would be necessary with an emulator such as mininet.

The space syntax of CaSL, developed to model discrete space in collective adaptive systems, provides a mechanism to describe a directed graph, and this separates the network topology information from that of component behaviour. In the SDN model, each switch is assigned a unique location and each host is assigned the location of the switch to which it is attached (multi-homed hosts can not be modelled currently). Each edge in the network graph is described by `[a] -> [b] { port = p }`. This specifies that there is a network connection from switch `a` to switch `b` and it is accessed through port `p` on switch `a`. Figure 5 illustrates how this language can be used to describe the left-most pod of Figure 1 and the nine hosts it supports.

We number the six switches in the left-most pod with 0,...,6 from left to right and top to bottom, and the six ports of a single switch are numbered in the same way. The nine hosts are numbered from 0 to 8. The bottom right switch of the pod is switch 5, with port 0 connected to switch 0, port 1 to switch 1 and port 2 to switch 2. Furthermore, port 3 is connected to host 6, port 4 to host 7 and port 5 to host 8. The ports 0, 1 and 2 of switches 0, 1 and 2 are not connected since we are considering the pod on its own for this example. Using this numbering of switches, ports and hosts, we can describe the network topology using the `space` keyword. Six locations are defined in the `nodes` section, one for each switch. In the `connections` section, links between switches are defined and labelled with port numbers. Only switch locations are defined, and two constant arrays are required to specify the location of each host in terms of the switch to which it is connected, and the port number of each host.

In the specification in Figure 5, the actual space specification is parametric and hence solely dependent on the value of `k`. By contrast, the constant definitions are specific for `k=6`. In the full SDN model, all of these are defined parametrically and hence a fat-tree topology of any size can be specified by changing the value of `k`.

The definition of the collective in Figure 6 defines the location of each switch and host component using the $@[\ldots]$ notation. Each switch is located at its switch location, and *Host_Switch* is used to determine the location of each host. In the current model, hosts are not mobile, and hence their locations are fixed. However, it would be straight-

forward to have mobile hosts whose locations (represented by the switch to which they are currently attached) change over time. In this case, the array would be used to define the initial location of each host.

5.2 Space and communication

All communication between switches and hosts use a single unicast action comm (we discuss communication between switches and controller below). The fact that a single action is used is exactly what allows for the definition of generic components. If comm were replaced with actions that describe communication between specific components, it would be necessary to describe each network element as a separate component. In many process-algebra-style languages, the use of a single action would mean that all process with that action would be able to communicate with each other (assuming various forms of hiding are not used). Because CARMA implements attribute-based communication, predicates can be used to ensure that this free-for-all does not happen. The predicates used in the SDN model use the space description to determine which network elements are directly connected to each other. These are captured by the functions appearing in Figure 3, specifically *ReceiverIsConn* and *SenderIsConn*. These functions take two arguments, my.*loc* and *loc*, which are the location of the sender and receiver respectively. In the case of communication between a switch and a host communication, it is necessary to check that the switch and host have the same location. In the case of communication between a switch and another switch, it is necessary to check that the receiver switch is in the post set of the sending switch with respect to the topology defined by the space description. This is checked with the syntax `loc in my.loc.post`. There are additional checks on port numbers which are defined on the connections.

By contrast, *ReceiverIsContr* and *SenderIsContr*, and the predicates $loc = [i]$ in the controller in Figure 4 just check that the identifier is correct rather than for any connectivity. This provides for different levels of abstraction of networking in the model. To focus on network performance between hosts, we have chosen to model the movement of packets in the network between hosts at a fine-grained level (but not so fine-grained that we model varying packet sizes) whereas the communication between switches and controller is modelled more abstractly without packet-level details. This choice allows a focus on specific aspects of the model; in our case, the measurement of latency of actual traffic in the network. For the two scenarios we are considering, the secondary traffic between switches and controller is only for data collection and plays a negligible role and hence we can omit its detailed modelling.

5.3 The collective and environment

Figure 6 describes global constants and variables, together with the initial specification of the collective and the evolution functions of the environment. The number of ports in a switch is a global constant as are the total number of switches and host. Additional aspects of the space description are global constants. Globally, two variables are tracked. The number of packets that have been received, together with total amount of time taken by these packets in traversing the next allowing the creation of a measure to describe this with the following syntax.

Global constants:	k	number of ports in a switch
	S	total number of switches: $S = 5k^2/4$
	H	total number of hosts: $H = k^3/4$
	FatTree	description of network structure
	Host_Switch	array that allocates hosts to switches
	Host_Port	array that allocates hosts to ports
Global store:	*pkt_time*	sum of time taken by each packet
	pkt_count	total number of packets received
Collective:	*Timer*	single Timer component
	$Switch@[i]$	for $0 \leq i \leq S-1$
	$Controller@[Con]$	single Controller component
	$Host@[Host\_Switch_j]$	for $0 \leq j \leq H-1$

Evolution rule functions:

$$\mu_p(\gamma_s, \gamma_r, \alpha) = 1$$

$$\mu_w(\gamma_s, \gamma_r, \alpha) = 1$$

$$\mu_r(\gamma_s, \alpha) = \begin{cases} \lambda_c & \alpha = \mathsf{comm} \\ \lambda_{ss} & \alpha = \mathsf{send\_stats} \\ \lambda_{ru} & \alpha = \mathsf{rule\_update} \\ \lambda_{cc} & \alpha = \mathsf{contr\_comm}^\star \\ \ldots & \alpha = \mathsf{gen\_pkt\_type}_i^\star \\ \ldots & \text{other traffic pattern actions} \\ fast & \text{otherwise} \end{cases}$$

$$\mu_u(\gamma_s, \alpha) = \begin{cases} \{pkt\_count \leftarrow pkt\_count + 1 \\ \quad pkt\_time \leftarrow pkt\_time + (\mathsf{now} - \gamma_s(in\_pkt.time\_sent)\}, 0 \\ \qquad \alpha = \mathsf{log\_packet}^\star \\ \{\}, 0 \qquad \text{otherwise} \end{cases}$$

Fig. 6. Global constants, global store, collective and environment functions

```
measure average_latency
                = global.pkt_time / global.pkt_count
```

As mentioned previously, the collective defines the individual copies of components with their locations. The timer component is not located. The evolution rule functions for probabilities and weights have a default value of 1 as these are not used in the SDN modelling. Most actions have the rate *fast* which means that they are essentially immediate. Other actions include communication, sending of statistics, updating of rules have constant rates, and as discussed previously traffic pattern rates may be based on information about senders and receivers.

The update function does nothing for all actions except $\mathsf{packet\_log}^\star$. In the case of this action, when a packet is received by any host, the packet count is increased and the total time taken is increased appropriately, using information about when the packet was sent which is stored in the packet itself. This abstracts from the details relating to clock drift that can be involved in measuring packet latency in real networks. None of the actions introduce new components to the collective. In the SDN model described

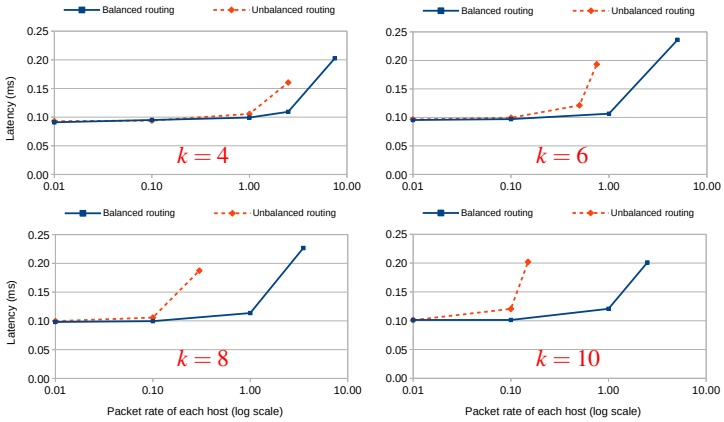


Fig. 7. Packet latency for balanced and unbalanced routing

here, there is a fixed number of network elements which neither increase nor decrease during the model execution. In a model where hosts may appear and disappear, the update function would specify the action that would trigger the introduction of new host which would be instantiated with the relevant attribute values. Host components can disappear through the use of **kill**. It is also possible to add behaviour to capture the temporary impairment of all network elements.

6 Experiments and results

We consider two experiments to illustrate the use of CARMA and MultiVeStA in considering the scalability of the fat-tree topology in the context of SDN and data centres [5]. These experiments report of the averages of measures and probabilities over multiple simulation runs performed by the CARMA software. We also mention the results of a security-focussed experiment that has been reported elsewhere [10]. These models assume the use of gigabit per second networks and have the communication and switch rates calibrated accordingly.

6.1 Scalability of topology

This experiment considers two important aspects of fat-tree topology. In the first place, it compares the effect of multiple routes and how this allows for higher traffic loads to be viable. This is achieved by considered a balanced routing over the network and an unbalanced routing where all routes go through a single core switch of the network. Furthermore, we consider these for different number of ports, to assess whether the scaling is similar for different size networks. For this scenario, we assume equal traffic between each pair of hosts, and increase the rate of this traffic to the point that the switches start to become overloaded, and the packet latency increases so that the switches are unable to support the line speed of the network. These results are reported in Figure 7. Data points are only included for experiments where latency reached steady state. The

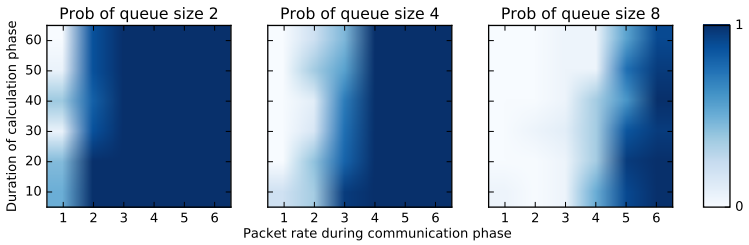


Fig. 8. Heat maps of queue size probabilities

fat-tree topology scales moderately well when routing is balanced. For k of 6 or more, performance starts to drop off at a rate of 1.00. In the case of $k = 8$, there are 16256 flows at this rate, and for $k = 10$, there are 62250 flows at this rate (the table in Figure 9 shows how the number of flows increase as k increases). In the case when routing is unbalanced to the extent that all flows go through a single core router, performance declines at lower rates as k increases. This emphasises the need for an SDN controller to utilise good load balancing algorithms to ensure good performance.

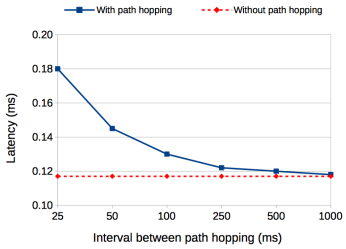
6.2 MapReduce traffic modelling

MapReduce describes a particular pattern of interaction between computers performing large computations [2]. A fixed number of hosts, say n, first perform a number of computations that can be done independently (the *map* phase). Once this phase is over, the results of the computation must be integrated via some computation and hence must be transferred to a single host (the *reduce* phase). This results in periods of limited traffic and then high traffic for the communication of results.

The SDN CARMA model supports the modelling of this traffic pattern. When this pattern is activated, then there is an exponentially distributed duration in which there is limited communication between hosts, followed by a duration (drawn from a different exponential distribution) during which there are high levels of traffic from $n-1$ of the hosts to the remaining host, and this alternating pattern repeats. We will refer to these as the *computation* phase and the *communication* phase, respectively.

In the previous experiment above that considered uniform traffic, it was assumed that the data centre network was being used by a single client (or alternatively that each host was shared by every client in the data centre) since there was traffic between all hosts. In the MapReduce experiment, a more constrained scenario is considered. First, for reasons of security, we assume that each host is used by a single client; second, we assume that each client has been allocated $2k^2/4$ hosts, and these hosts are connected to two pods. Thus, some of the traffic in the communication phase must traverse core switches to move between pods. For a fat-tree topology-based on k-port switches, this means that there are $k/2$ clients, each with a MapReduce traffic pattern (using the same exponential rates for periods of computation and communication).

We are interested in understanding the effect on maximum queue size of different rates for the MapReduce traffic pattern. We use a MultiVeStA query to determine the probabilities of a particular queue size for variations in the duration of the computation



k	4	6	8	10
1 tu	0.2s	1.3s	5.2s	19.4s
1000 tu	0.1h	0.4h	1.4h	5.4h
Hosts	16	54	128	250
Flows	240	2862	16256	62250
Packets	77.2K	132.5K	126.3K	248.5K
Packet rate	5.0	2.5	1.0	1.0
Components	38	101	210	377

Fig. 9. Packet latency for path hopping mitigation of the Sneak-Peek attack [10] (left). Execution times for different values of k (right).

phase and variations in rate of traffic during the communication phase. We do not vary the duration of the communication phase. Figure 8 illustrates the probabilities obtained when considering a queue size of 2, 4 and 8. As can be seen, the rate of traffic during the communication phase has a greater impact on the queue size than the average duration of the gap (expresses as milliseconds) for the range of value considered. Since larger queue sizes (over 2) are associated with increased packet latency, these results demonstrate that decreased performance is likely to occur with packet rates over 4 for communication duration of up to 60ms.

6.3 Trade-offs in network security

We have also investigated security-performance trade-offs for the fat-tree topology [10] when considering mitigations of a covert channel attack call Sneak-Peek [41]. The mitigations involves changing route frequently (called *path hopping*) to disrupt an existing covert channel. Path hopping requires that new routes are sent to switches which causes delays in packet processing and increases latency. Realistic data for switch update delays have been used [37]. Experimentation (shown in Figure 9) revealed that very frequent path hopping (every 25ms) results in a 50% increase in latency. However, path hopping every second did not appear to increase the latency. Since the ability to create a covert channel with the Sneak-Peek attack requires shared network infrastructure, this model has been further investigated with MultiVeStA queries of the form "What is the probability that there will be a shared network component for a duration of x milliseconds?" thereby quantifying the risk of an attack [10].

7 Evaluation and conclusion

We now evaluate the use of CARMA and MultiVeStA and consider some metrics of relevance. We compare various metrics for the uniform traffic experiment described in Section 6.1 and the figures in right-hand-side of Figure 9 were obtained on a single core of a 2.66Hz processor with 8GB of memory available. Each time unit (tu) of simulation represents a millisecond of network time. The packet rate was chosen to be at a level where the network was fully loaded but not congested. The table includes the number of flows (between each pair of hosts), the packet rate and the number of packets transmitted

in 1000tu. The last line is the number of components in the CARMA model. The time for the simulation of almost 20 seconds of network time in the 10-port case is moderately high; however, many independent simulations can be spread over different cores or computers, and hence averages over all simulations can be obtained in that time.

In terms of model construction, the fat-tree size can be increased by changing a single constant in the model code. Traffic patterns such as the MapReduce pattern can be specified parametrically as well. Hence it is possible to investigate regular topologies by only changing the model for the topology and relevant traffic patterns. Irregular network structures are likely to require more modifications of the mode code. The current model is around 1200 lines of CaSL code. Of this, about 25% of the code is components, collective, environment and measures; 10% is network and traffic specification; and 10% is parameters and data structure definition. The remainder are the function definitions required for the model, including the topology and routing determination, the predicates for constraining interaction between components, and calculations of network flows.

Further research includes comparison of model-building time and effort needed for the CARMA approach and the use of mininet, both for the initial model and for model changes; validation of small-scale CARMA models using mininet emulation, and experimentation with data and configurations from a real datacentre.

This research has demonstrated that CARMA is indeed suitable for modelling network and there is a good match between the discrete space syntax and the requirement for the description of network topology. Furthermore, this supports generic component definition, and leads to a model with a few generic components which aids comprehension. The richness of CARMA in terms of data structures and support for functions provides a step beyond what is usually possible in process-algebra-style modelling.

The comparison of the usability of our approach to others is hard to assess, partly because of different levels of abstraction. Certainly, our approach provides a level of abstraction that is novel in terms of assessing the performance of SDN networks. Furthermore, if we know that the generic components work correctly, then debugging can focus on the network specification. Hence, this level of abstraction allows for speed in developing, debugging and simulating models with a fast turn-around which is not possible with full-stack emulation or simulation methods, as shown by similar formal methods research [31]. Furthermore, very few formal methods take a quantitative approach which is required if performance is to be measured. The hybrid simulation approach using DEVS and TopoGen [8, 30] is the closest to our research and works with network specification languages that could be used to obtain the CARMA network description. We believe that our style of light-weight modelling style has a role to play, and the next phase of development is to embody our approach in software that conceals the details of the CARMA from a user without experience of formal methods to provide a software tool which enables specification of network topologies and traffic patterns in a simple format with graphical elements where appropriate.

Acknowledgements This work is supported by the EPSRC project Robustness-as- Evolvability, EP/L02277X/1 and the EPSRC Platform Grant EP/N014758/1. Thanks to David Aspinall, Wei Chen and Henry Clausen for their useful comments. The CARMA models and experimental data can be obtained from `groups.inf.ed.ac.uk/security/RasE/`.

References

1. Abd Alrahman, Y., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: Proceedings of FORTE 2016. pp. 1–18 (2016)
2. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. In: Proceedings of the ACM SIGCOMM 2008. pp. 63–74 (2008)
3. Al-Shaer, E., Al-Haj, S.: FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures. In: Proceedings of SafeConfig '10. pp. 37–44 (2010)
4. Anderson, C., Foster, N., Guha, A., Jeannin, J.B., Kozen, D., Schlesinger, C., Walker, D.: Netkat: Semantic foundations for networks. SIGPLAN Notices 49, 113–126 (2014)
5. Bilal, K., Khan, S., Zhang, L., Li, H., Hayat, K., Madani, S., Min-Allah, N., Wang, L., Chen, D., Iqbal, M., Xu, C.Z., Zomaya, A.: Quantitative comparisons of the state-of-the-art data center architectures. Concurrency and Computation: Practice and Experience 25, 1771–1783 (2013)
6. Calheiros, R., Ranjan, R., Beloglazov, A., Rose, C.D., Buyya, R.: CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software Practice and Experience 41, 23–50 (2011)
7. Canini, M., Venzano, D., Perešíni, P., Kostić, D., Rexford, J.: A NICE Way to Test OpenFlow Applications. In: Proceedings of NSDI 2012 (2012)
8. Castro, R., Kofman, E.: An integrative approach for hybrid modeling, simulation and control of data networks based on the {DEVS} formalism. In: Modeling and Simulation of Computer Networks and Systems, pp. 505 – 551. Morgan Kaufmann (2015)
9. Chaves, L., Garcia, I., Madeira, E.: Ofswitch13: Enhancing ns-3 with OpenFlow 1.3 support. In: Proceedings of WNS3 '16. pp. 33–40 (2016)
10. Chen, W., Lin, Y., Galpin, V., Nigam, V., Lee, M., Aspinall, D.: Formal analysis of SneakPeek: A data centre attack and its mitigations (2018), IFIP SEC 2018, to appear
11. Ciocchetta, F., Hillston, J.: Bio-PEPA for epidemiological models. Electronic Notes in Theoretical Computer Science 261, 43–69 (2010)
12. Dacier, M.C., Dietrich, S., Kargl, F., Knig, H.: Network Attack Detection and Defense (Dagstuhl Seminar 16361). Dagstuhl Reports 6(9), 1–28 (2017)
13. De Nicola, R., Latella, D., Loreti, M., Massink, M.: A uniform definition of stochastic process calculi. ACM Computing Surveys 46, 5 (2013)
14. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM TAAS 9, 7:1–7:29 (2014)
15. Farhadi, H., Lee, H., Nakao, A.: Software-defined networking: A survey. Computer Networks 81, 79–95 (2015)
16. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.: A process algebra for wireless mesh networks. In: Proceedings of ESOP 2012. pp. 295–315 (2012)
17. Fernandes, S.: Performance Evaluation for Network Services, Systems and Protocols. Springer (2017)
18. Galpin, V.: Modelling network performance with a spatial stochastic process algebra. In: Proceedings of AINA 2009. pp. 41–49 (2009)
19. Galpin, V.: Modelling ambulance deployment with CARMA. In: Proceedings of COORDINATION 2016. pp. 121–137 (2016)
20. Galpin, V., Georgoulas, A., Gilmore, S., Hillston, J., Latella, D., Loreti, M., Massink, M., Zoń, N.: CASL at work. QUANTICOL Deliverable D4.3 (2017), http://blog.inf.ed.ac.uk/quanticol/deliverables
21. Galpin, V., Zon, N., Wilsdorf, P., Gilmore, S.: Mesoscopic modelling of pedestrian movement using carma and its tools. ACM TOMACS 28, 11:1–11:26 (2018)

22. Heller, B., Sherwood, R., McKeown, N.: The controller placement problem. In: Proceedings of HotSDN '12. pp. 7–12 (2012)
23. Hillston, J.: A compositional approach to performance modelling. CUP (1996)
24. Hillston, J., Loreti, M.: Specification and analysis of open-ended systems with CARMA. In: Proceedings of E4MAS 2014. pp. 95–116. Springer (2015)
25. Hillston, J., Loreti, M.: CARMA Eclipse Plug-in: A tool supporting design and analysis of collective adaptive systems. In: Proceedings of QEST 2016. pp. 167–171 (2016)
26. Ivey, J., Yang, H., Zhang, C., Riley, G.: Comparing a scalable SDN simulation framework built on ns-3 and DCE with existing SDN simulators and emulators. In: Proceedings of SIGSIM-PADS 2016. pp. 153–164 (2016)
27. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.: VeriFlow: Verifying Network-Wide Invariants in Real Time. In: Proceedings of NSDI 2013 (2013)
28. Kouzapas, D., Philippou, A.: A process calculus for dynamic networks. In: Proceedings of FMOODS and FORTE 2011. pp. 213–227 (2011)
29. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: Rapid prototyping for software-defined networks. In: Proceedings of Hotnets-IX. pp. 19:1–19:6 (2010)
30. Laurito, A., Bonaventura, M., Astigarraga, M., Castro, R.: TopoGen: A network topology generation architecture with application to automating simulations of software defined networks. In: Proceedings of WSC 2017. pp. 1049–1060 (2017)
31. Lemos, M., Dantas, Y., Fonseca, I., Nigam, V.: On the accuracy of formal verification of selective defenses for TDoS attacks. Journal of Logical and Algebraic Methods in Programming 94, 45–67 (2018)
32. Loreti, M., Hillston, J.: Modelling and analysis of collective adaptive systems with CARMA and its tools. In: Proceedings of SFM 2016. pp. 83–119 (2016)
33. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P., King, S.: Debugging the Data Plane with Anteater. In: Proceedings of SIGCOMM '11 (2011)
34. Majumdar, R., Tetali, S., Wang, Z.: Kuai: A model checker for software-defined networks. In: Proceedings of FMCAD '14. pp. 27:163–27:170 (2014)
35. Pascoal, T., Dantas, Y., Fonseca, I., Nigam, V.: Slow TCAM Exhaustion DDoS Attack. In: Proceedings of SEC 2017. pp. 17–31 (2017)
36. Prakash, C., Lee, J., Turner, Y., Kang, J.M., Akella, A., Banerjee, S., Clark, C., Ma, Y., Sharma, P., Zhang, Y.: PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In: Proceedings of SIGCOMM '15. pp. 29–42 (2015)
37. Rotsos, C., Sarrar, N., Uhlig, S., Sherwood, R., Moore, A.: OFLOPS: an open framework for openflow switch evaluation. In: Proceedings of PAM 2012 (2012)
38. Sebastio, S., Vandin, A.: MultiVeStA: statistical model checking for discrete event simulators. In: Proceedings of ValueTools '13. pp. 310–315 (2013)
39. Smolka, S., Kumar, P., Foster, N., Kozen, D., Silva, A.: Cantor meets Scott: semantic foundations for probabilistic networks. In: Proceedings of POPL 2017. pp. 557–571 (2017)
40. Son, S., Shin, S., Yegneswaran, V., Porras, P., Gu, G.: Model checking invariant security properties in OpenFlow. In: Proceedings of IEEE ICC 2013. pp. 1974–1979 (Jun 2013)
41. Tahir, R., Khan, M.T., Gong, X., Ahmed, A., Ghassami, A., Kazmi, H., Caesar, M., Zaffar, F., Kiyavash, N.: Sneak-peek: High speed covert channels in data center networks. In: Proceedings of IEEE INFOCOM 2016. pp. 1–9 (2016)
42. Zhou, W., Jin, D., Croft, J., Caesar, M., Godfrey, P.: Enforcing customizable consistency properties in software-defined networks. In: Proceedings of NSDI 2015 (2015)
43. Zoń, N., Gilmore, S., Hillston, J.: Rigorous graphical modelling of movement in collective adaptive systems. In: Proceedings of ISoLA 2016 (2016)